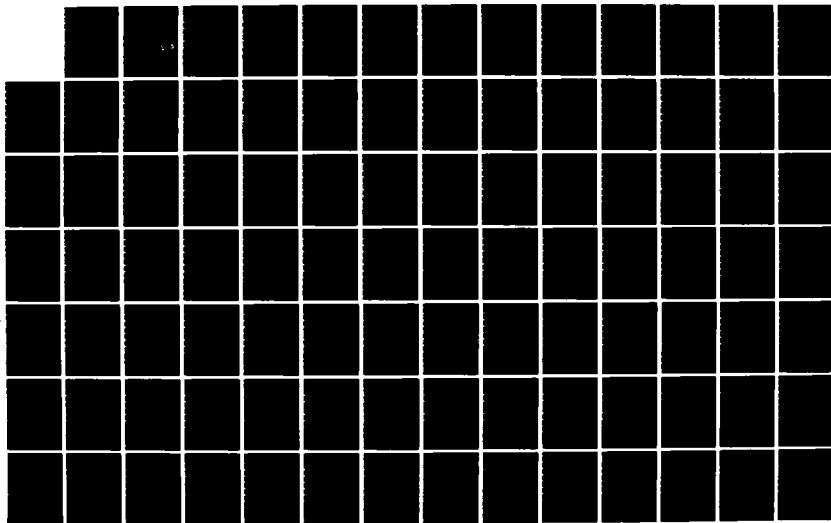
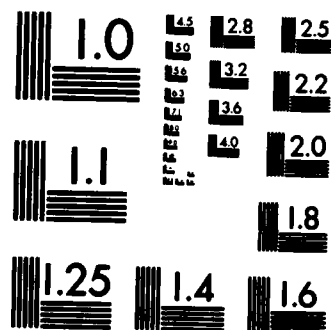


AD-A142 831 HIGH SPEED LOW-COST WAYS TO GET MESSAGES FROM A SENDER 1/2
TO A RECEIVER WHEN (U) VLYK LTD ANN ARBOR MI
B BLAKLEY 28 MAY 84 VLYK/AFOSR/SBIRI/83-84/001
UNCLASSIFIED AFOSR-TR-84-0528 F49620-83-C-0160 F/G 17/2.1 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

4

AD-A142 831

FINAL REPORT ON
"HIGH-SPEED LOW-COST WAYS TO GET MESSAGES FROM A SENDER TO A RECEIVER
WHEN SOME CHANNELS LINKING THEM BECOME INOPERATIVE."

DTIC
ELECTE
JUL 9 1984
S A D

DTIC FILE COPY

Approved for public release;
distribution unlimited.

REPORT DOCUMENTATION PAGE

UNCLASSIFIED		Approved for Public Release; Distribution Unlimited	
MONITORING ORGANIZATION REPORT NUMBER NLYK/AFOSR/SBIR/83-84/001		MONITORING ORGANIZATION REPORT NUMBER AFOSR-TR- 84-0328	
1. NAME OF MONITORING ORGANIZATION NLYK Ltd.	2. OFFICE SYMBOL Hq. AFOSR	3. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Research	
4. ADDRESS (City, State and ZIP Code) 2440 Stone Ann Arbor, MI 48105		5. ADDRESS (City, State and ZIP Code) Directorate for Mathematical and Informational Sciences Bolling AFB, DC 20332	
6. NAME OF FUNDING SPONSORING AGENCY AFOSR	7. OFFICE SYMBOL NM	8. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F49620-83-C-0160	
9. ADDRESS (City, State and ZIP Code) Bolling AFB, DC 20332		10. SOURCE OF FUNDING NOS.	
		PROGRAM ELEMENT NO. 61102F	PROJECT NO. 3005
		TASK NO. A1	WORK UNIT NO.
11. SUBJECT TERMS (Continue on reverse side if needed and identify by block number) Science, Engineering, AFOSR SBIR, Electronic Technology, Electronic Component Reliability, Digital Communications. (See continuation on reverse side.)			
12. AUTHOR (Last Name, First Name, Middle Initial) Blakley, Bob			
13. DATE COVERED From 83 Sep 30 to 84 Mar 31		14. DATE OF REPORT (Yr, Mo, Day) 84 May 28	
15. TYPE OF REPORT Final		16. NUMBER OF PAGES 150	

This AFOSR SBIR Phase 1 Project produced explicitly the hyper-fast pool/split/recombine algorithms of the Bloom technique. These algorithms, once they obtain hardware implementation, will be used as follows. They will make it possible for a sender to send all desired digital information to a receiver by coding it for transmission over several parallel channels in such a way that decoding will recover everything sent even when up to a predetermined number of channels fail. This project developed a set of designs and plans for hardware implementation of such p/s/r processes by means of existing microprocessors of sizes 4, 8 and 16 bits.

Dr. Robert Wood, AFOSR

760-5000

CONTINUATION OF BOX 10, FROM PREVIOUS PAGE:

CS, Computer Science, Jamming, Antijam, Red Noise, Mathematics, Information Theory, Error Control Codes, Threshold Schemes, Hyperfast Bloom Processes, Pool/split/restitute Processes, Algorithms, Finite Fields, Galois Fields, Vandermonde Matrices, Reed-Solomon Codes, Vector Spaces, Linear Transformations, Gate Array, Programmable Logic Array, Parallel Processing, Microprocessors, Packet Switching, Chip Design, Fiber Optic Communications, Spread Spectrum, Message Gap, Distributed Processing, Netted Communications Systems, VLSI Design, Reliability, Survivability, Endurability, Responsiveness, Space Systems Technology, Distributed Communications, Radio Frequency Communications, High Bandwidth Communications, Channel Failure, Channel Fading, Sending Node, Receiving Node, Bandwidth Expansion, Novel Architecture, Fault Tolerance, Fail Safe Communications, Efficient Spectrum Utilization, Cost Effectiveness, Telecommunications Networks, Local Area Networks, Low Computational Complexity, Encode, Decode, Redundancy, Minimum Redundancy, Parallelism, Parallel Computations, Systolic Processors, Minimum Message Expansion.

BOX 11. TITLE (Include Security Classification)

UNCLASSIFIED: FINAL REPORT ON "High-speed low-cost ways to get messages from a sender to a receiver when some channels linking them become inoperative."

Chief, Bureau of Information Division

The work described in the YLYK Ltd. proposal (see Appendix A) which led to this SBIR contract was completed on time, and within budget. Moreover the outcome was largely definitive and was at least as good as the target outcome of the proposal. This report is prepared to meet the 84 May 31 deadline for final report. In summary, the work was carried out on time, on target, within budget. This report is timely.

0. Introduction

- 0710
COPY
INSPECTED
2

Section 3, "Summary of tasks, work, discoveries, recommendations and alternatives" is the heart of the report. It describes how YLYK Ltd. performed its agreed-upon Task 1 and Task 2. The reader may want to skim it before going through the report as a whole.

1. Overview Narrative

1.1 Red Noise

Appendix A contains a copy of the YLYK Ltd. proposal which led to the contract on which this is the final report. In the interests of readability we restate the idea behind the p/s/r processes, along with some realistic instances.

A sender S and a receiver R are linked by n channels of approximately equal capacity. All communications are digital, i.e. are strings of bits (0 or 1). The sender and the receiver anticipate traumas which will inactivate some of these channels. Nevertheless both the sender and the receiver expect at least k of the n channels to continue to function. Here, as everywhere, it is assumed that $k \leq n$.

They face the "red noise" problem. How does the sender S encode k channels worth of information for sending along n channels to the receiver R in such a way that R can recover all the information cheaply and quickly as long as any k of the n channels remain operative? The sender S must encode in ignorance of which k channels will survive the trauma and remain operative. Examples of the red noise problem are numerous. We sketch out a few here. We will return to them.

i. On-chip. Certain elements on a chip may fail permanently. The number n of channels is typically less than 100, often less than 10. Here k is usually almost as big as n , since chips with lots of hard failures are typically discarded. Perhaps $k = n - 1$ is especially important.

ii. Packet switching. Here the packets are the "channels". Occasionally a packet is destroyed or irrevocably misrouted. The number n of total packets for many practical examples would be less than 200, often less than 20. Most packets should arrive intact, so k would usually be near n . Perhaps $k = n - 1$ is especially important.

iii. Spread spectrum. Here a "channel" might be a frequency if the technique employed is frequency hopping. Perhaps quite a few frequencies are jammed. The number n of total frequencies should usually be less than 60,000 and often considerably less than 4,000. k can vary all over the lot. In battle conditions we might have $k < n/10$, e.g. only $k = 70$ "clean" frequencies among $n = 1,000$ frequencies being used. Those who feel that this is a pessimistic estimate should consult McEliece's recent paper on jamming in Longo's Springer-Verlag book, Secure Digital Communications.

iv. Hard wires or fibers. A control center on a weapons platform (plane, ship, etc.) might be connected by $n = 30$ parallel fibers to a propulsion unit, sensor, control surface, or weapons pod. It might be desirable to maintain full communication even after 20 fibers were cut. Here $k = 10 = n/3$. In such examples n less than 200 seems plausible. k can vary all over the lot.

v. Multiple channels between manned centers. A city might talk to a command post via a mixture of twisted pairs, fibers, microwave relay paths and satellite links. It would be desirable to keep up communication if half of the $n = 20$ channels joining them fail.

In all the foregoing examples the number n of total channels before failures occur would satisfy the inequality

$$2 \uparrow 0 = 1 \leq n \leq 65,536 = 2 \uparrow 16$$

(where we use the ALGOL arrow notation $2 \uparrow 16$ instead of the older exponent notation 2^{16}). We will adopt the inequality above once for all as an explicit assumption :

At least one "channel";

At most 65,536 "channels".

The reader is asked to bear it in mind everywhere below. Another categorical assumption is:

Every signal is digital.

1.2 Bloom pool/split/restitute processes

A solution to the red noise problem is called a p/s/r process. We will discuss only Bloom p/s/r processes and their close relatives here. See Appendix G for the first exposition of the idea behind Bloom threshold schemes and p/s/r processes. They make use of many of the ideas which arise in Reed-Solomon error control codes. But we will not explicitly pursue any resemblances to the latter structure.

The idea behind a k-out-of-n Bloom p/s/r process is to enable a sender to use finite field arithmetic and linear algebra to smear k channels worth of information into n channels worth of transmission to a receiver R in such a way that all the original information can be quickly reclaimed from the outputs of any k of the n channels, even if n - k of them do not carry any information to the receiver (i.e. even if n - k of the n channels are inoperative).

Bloom's approach to building a k-out-of-n p/s/r process makes use of a field F containing at least n elements, and a k dimensional vector space V over F. It is easy to verify that there is at least one collection

$$B = \{B(1), B(2), \dots, B(n)\} \subseteq V$$

of n vectors in general position in V (meaning that every k-member subset of B is a basis of V). Sender S and receiver R agree on one such B and refer everything to it. Given a list

$$I = (I(1), I(2), \dots, I(k)) \subseteq F^k$$

of k pieces of information (i.e. k members of the field F) define a linear functional

$$L: V \rightarrow F$$

with the property that

$$L(B(j)) = I(j)$$

for each positive integer $j \leq k$. These k pieces of information provide a complete unique specification of the linear map L since

$$\{B(1), B(2), \dots, B(k)\}$$

is a basis of V . But

$$\{B(w(1)), B(w(2)), \dots, B(w(k))\}$$

is also a basis of V for any injection (one-to-one function)

$$w: \{1, 2, \dots, k\} \rightarrow \{1, 2, \dots, n\}$$

So you can reconstruct L , and therefore determine the list

$$I = (I(1), I(2), \dots, I(k))$$

if you know the value of L at any k members

$$B(w(1)), B(w(2)), \dots, B(w(k))$$

of the set V .

Now it is obvious how to encode and decode. To encode the list I , form L and send $L(B(j))$ down channel j for each positive integer $j \leq n$. To decode (i.e. to recover I from the signals received on any k of the n channels) form L and then determine

$$L(B(j)) = I(j)$$

for each positive integer $j \leq k$. This is possible since any $B(w(1)), B(w(2)), \dots, B(w(k))$ make up a basis for V , and since a linear map L with domain V is determined by its values on a basis of V .

1.3 Making hyperfast Bloom p/s/r processes. Stages.

YLYK Ltd. set out to take this simple mathematical structure, the abstract Bloom p/s/r process, and produce an abstract design of a p/s/r process which would run very fast on very cheap hardware. In this Phase I SBIR effort no attempt was made to produce or design hardware. Rather, the purpose of the work was to produce an abstract design of a

system capable of operating at megabit per second rates and above. On the basis of this abstract design the hardware design should be possible with few or no further abstract considerations.

Roughly speaking, the problems to be overcome fall into 4 stages:

1. Cold precomputation. The cold precomputation must be done before the p/s/r equipment is built. These precomputations will not slow down system operation. It would be perfectly acceptable if they took months to perform. In fact they can be completed in a minute except in very large cases discussed below.
2. Cool precomputation. The cool precomputations take place each time sender S and receiver R agree on the k and the n for a session of communication using a k-out-of-n p/s/r process. The cool precomputations will involve a minor delay, probably causing no inconvenience. This delay will usually be less than a second in reasonable sized cases as noted below.
3. Hot precomputation. The hot precomputation takes place after some channels have gone down. The receiver determines which k channels are still operating. This amounts to finding out which subset $B(w(1)), B(w(2)), \dots, B(w(k))$ of B will be used. Since the communication session is ongoing, any delay here is undesirable. Either you lose information on the fly or you pay for a buffer to hold undecoded material until your decode goes on stream. Unfortunately the hot precomputations can take many milliseconds. It is doubtful that a significant further improvement over the scheme YLYK Ltd. has formulated is possible here.
4. Real-time on-line encode or decode. The real-time on-line encode or decode stage should be able to keep up with high bit rate inputs. In an "impedance matched" situation the computer clock should tick at least once per arriving bit. For example, consider a 5-out-of-9 p/s/r process. Suppose that each of the 5 operative channels carries a signal at 10 megabits per second and that the "matched computer clocks" in the decoding system therefore push the computer to perform 10 similar logical operations (such as XOR,

i.e. exclusive or, of 4-bit words) per microsecond. It would be desirable to produce decoded output on all 5 decoded plaintext channels at a rate of 10 megabits per second. It appears possible to achieve such throughput rates, but with a certain short lag time. For example, the 10 megabit per second decoded output might lag the received bit stream by 2 microseconds. In other words the decoded bit streams proceeds at the same rate as the received encoded bit streams. But the decoded streams lag the received encoded streams by a phase lag of 20 clock ticks, i.e. by 20 bits.

We must deal with each of these four computational stages separately. The first, the cold precomputation stage, is completely noncritical. Neither time nor memory is important as long as the needed output can be produced within months and does not consist of too many computer words. The second stage, the cool precomputation, is not very critical. Presumably it occurs in tranquil conditions while the sender S and the receiver R are agreeing on a k-out-of-n scheme. Days could elapse between the choice of k and n, and the time transmission starts. And almost always seconds will elapse. It is therefore unlikely that the procedure described below for cool precomputation will delay timely receipt of transmitted messages. Stage 3, the hot precompute, is usually the most critical. If it should take a second or more, one must decide whether to lose a lot of bits or spend money on buffers. Stage 3, therefore, requires extremely close attention. Stage 4, the real-time on-line decode, is crucial but not troublesome. There are ways to carry Stage 4 out at very high bit rates, given adequate hardware. There is a "phase lag" i.e. a lag of several bits between received input signal and decoded final signal. This lag can be reduced to a few microseconds in existing TTL logic. But reducing it to zero is an impossibility.

1.4. Making hyperfast Bloom p/s/r processes. Extreme cases of parameter settings.

So much for the four stages of computation. We turn now to parameter settings. How sensitive is a k-out-of-n p/s/r process to k and to n?

First let us dispense with the four extreme cases. These are the two trivial cases $k = 0$ or $k = n$ and the two easy but not completely trivial cases $k = 1$ or $k = n-1$. A 0-out-of- n p/s/r process is silly. No information sent on n channels produces no information received. No p/s/r coding is required. The n -out-of- n case is far from silly. It is the present state of affairs. Send a different message on each of n channels and hope they all get through. No p/s/r coding is required. The 1-out-of- n case is also easy to deal with without p/s/r coding. Send the same message on all channels and hope that at least one channel remains operative.

The $(n-1)$ -out-of- n case is more interesting. It will also be important in some applications. Synchronize the channels. To p/s/r encode the information let the first $n-1$ channels transmit their messages unaltered. But at each time t , add (modulo 2) the bits on the first $n-1$ channels and send this sum (it, too, will be a bit) on the n th channel. To decode when one of the first $n-1$ channels, say the j th, fails you do as follows. If $i \in \{1, 2, \dots, n\} \setminus \{j\}$ the decode transformation is the identity. The channel is carrying its message unaltered. But if $i = j$, just form the sum of the bits on channels $1, 2, \dots, j-1, j+1, \dots, n$. This will be what the j th channel would have carried if it were still operative. Note that the cold precomputation, cool precomputation and hot precomputation are nonexistent. The on line computation acts on a single bit from each channel. And, if implemented by fast hardware as indicated in Figure 1.4.1 below in the 7-out-of-8 case, the output bit rate is the same as the input bit rate, but with a lag of $3 = \log(8)$ bits (All logarithms in this report are the information theorist's logarithm to base 2).

For the first time we note a point which will be addressed more fully below. Encoding is a do-nothing operation on all plaintext channels (i.e. the first $n-1 = 7$ channels), and all plaintext channels remain synchronized. Encoding is a do-something operation on the 8th = n th channel. To keep all eight channels synchronized, the receiver must do something to every channel. In the 7-out-of-8 case this means 3 successive stages of adding 0 to what comes over every one of the first $n-1 = 7$ channels. A similar statement holds regarding the decode process.

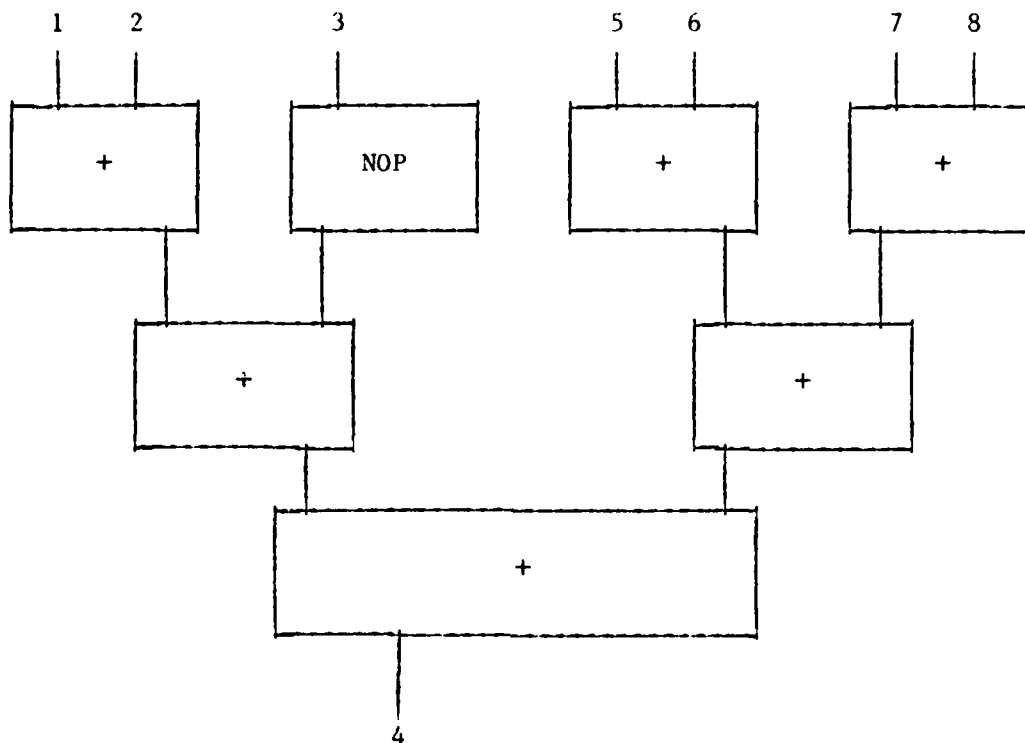


Figure 1.4.1.

The 7-out-of-8 p/s/r decode when channel 4 is inoperative. Assuming the modulo 2 adders (XOR) can operate as fast as bits are received the output bit stream will have the same speed as the input bit streams but will lag them by 3 bit positions. NOP means no operation. + stands for modulo 2 addition. Information flows downward.

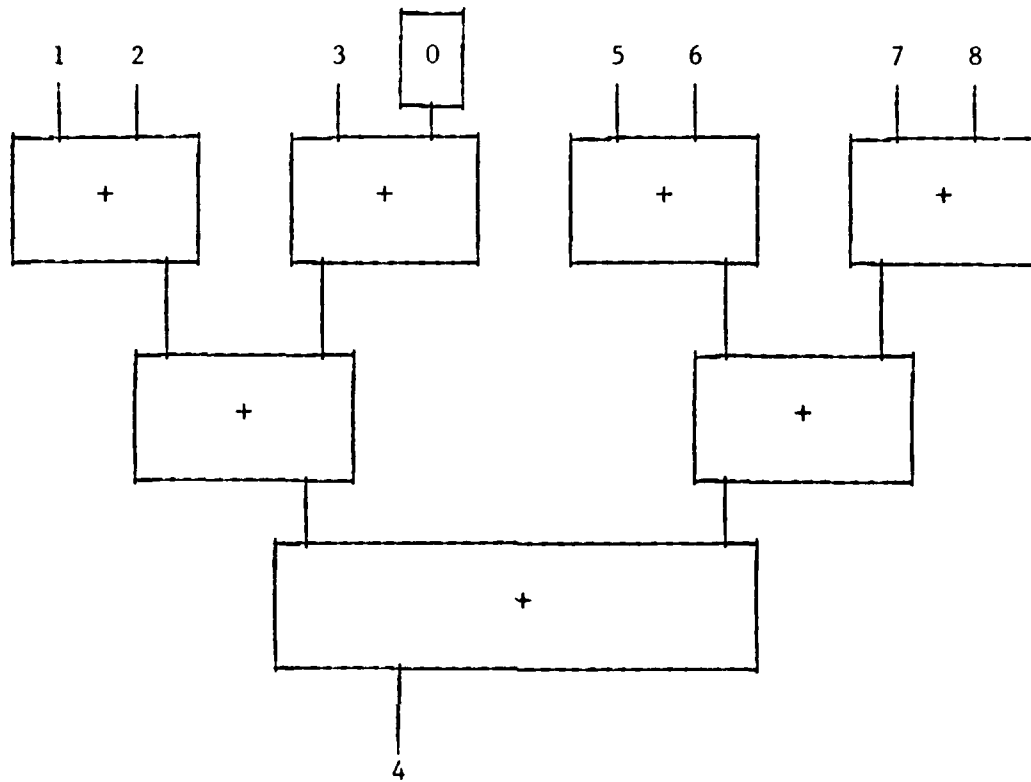


Figure 1.4.2

A variant of Figure 1.4.1. The receiver sends zeros into the decoder input corresponding to the missing channel 4. Information flows downward.

In each of the four extreme cases described in this subsection, the decode process could content itself with treating one bit at a time from each of the received channels. This is independent of n . Thus a very cheap programmable logic array (PLA) implementation of a hyperfast Bloom $(n-1)$ -out-of- n p/s/r process is possible for very large n . The lag time would be about $\log(n)$.

1.5 Making hyperfast Bloom p/s/r processes. Mean parameter settings.

Turning now from the four extreme cases to all the other cases, which we shall call mean cases, we note that the p/s/r processes we are dealing with always satisfy the inequalities

$$2 \leq k \leq n - 2 \leq n \leq 65,536 .$$

No mean p/s/r encode or decode can deal with just one bit at a time. In fact one must deal with "words" of length at least $\log(n)$ from each channel. Recall that all logarithms are the information theorist's logarithm to base 2 in this report. As noted in the YLYK Ltd. proposal to Air Force for this Phase I SBIR proposal, encode and decode will be done using $GF(2^Q)$ arithmetic. As noted above, we will deal only with $Q \leq 16$. We have already discussed the extreme $(n-1)$ -out-of- n case. This extreme case can be dealt with using $GF(2)$ arithmetic. In dealing with mean cases we will usually make the following assumption:

$$Q \in \{4, 8, 12, 16\}.$$

Thus we will often deal just with the arithmetic of $GF(16)$, $GF(256)$, and $GF(65,536)$. The reason for this is that 4, 8 and 16 bit words are natural objects to manipulate on standard hardware.

A case could be made for using only $GF(65,536)$, i.e. for sticking to 16 bit words for standardization, since such an implementation can "do everything". But this size seems unwieldy at present. It may be better to try to get as much mileage as possible out of the $GF(256)$ case, i.e. to try to get by with at most 256 transmitted channels. We will discuss some pros and cons later.

1.6 Making hyperfast Bloom p/s/r processes. Stage 4. Real-time on-line encode or decode.

In the mean cases of parameter settings one thing that does not change with parameter setting is the nature of the real-time on-line encode or decode in a superfast Bloom k-out-of-n p/s/r process. It is matrix multiplication. Encode is so like decode that we will concentrate on the latter in this section.

To each k-element subset

$$B^* = \{B(w(1)), B(w(2)), \dots, B(w(k))\}$$

of

$$B = \{B(1), B(2), \dots, B(n)\}$$

there corresponds a k by k matrix $DEC[B^*]$ such that

$$B(i) = \sum DEC[B^*](i,j)B(w(j))$$

for every positive integer $i \leq k$. The sum is over all positive integers $j \leq k$. As long as a given collection

$$\{w(1), w(2), \dots, w(k)\}$$

of channels is operative this square matrix $DEC[B^*]$ is unchanging. So the block diagram for decoding the ith channel is contained in Figure 1.6.1 below. For illustrative purposes Figure 1.6.1 describes a 7-out-of-25 Bloom p/s/r process in which the receiver knows that channels 1, 2, 5, 7, 10, 12 and 19 are operative. Since $25 \leq 32 = 2^5$ we can use 5-bit words, i.e. GF(32) arithmetic. So the 7 inputs to the decoder at time t are WORD1 on channel 1, WORD2 on channel 2, WORD5 on channel 5, ..., WORD19 on channel 19. Here of course

$$\begin{aligned} w(1) &= 1 \\ w(2) &= 2 \\ w(3) &= 5 \\ &\vdots \\ w(7) &= 19 \end{aligned}$$

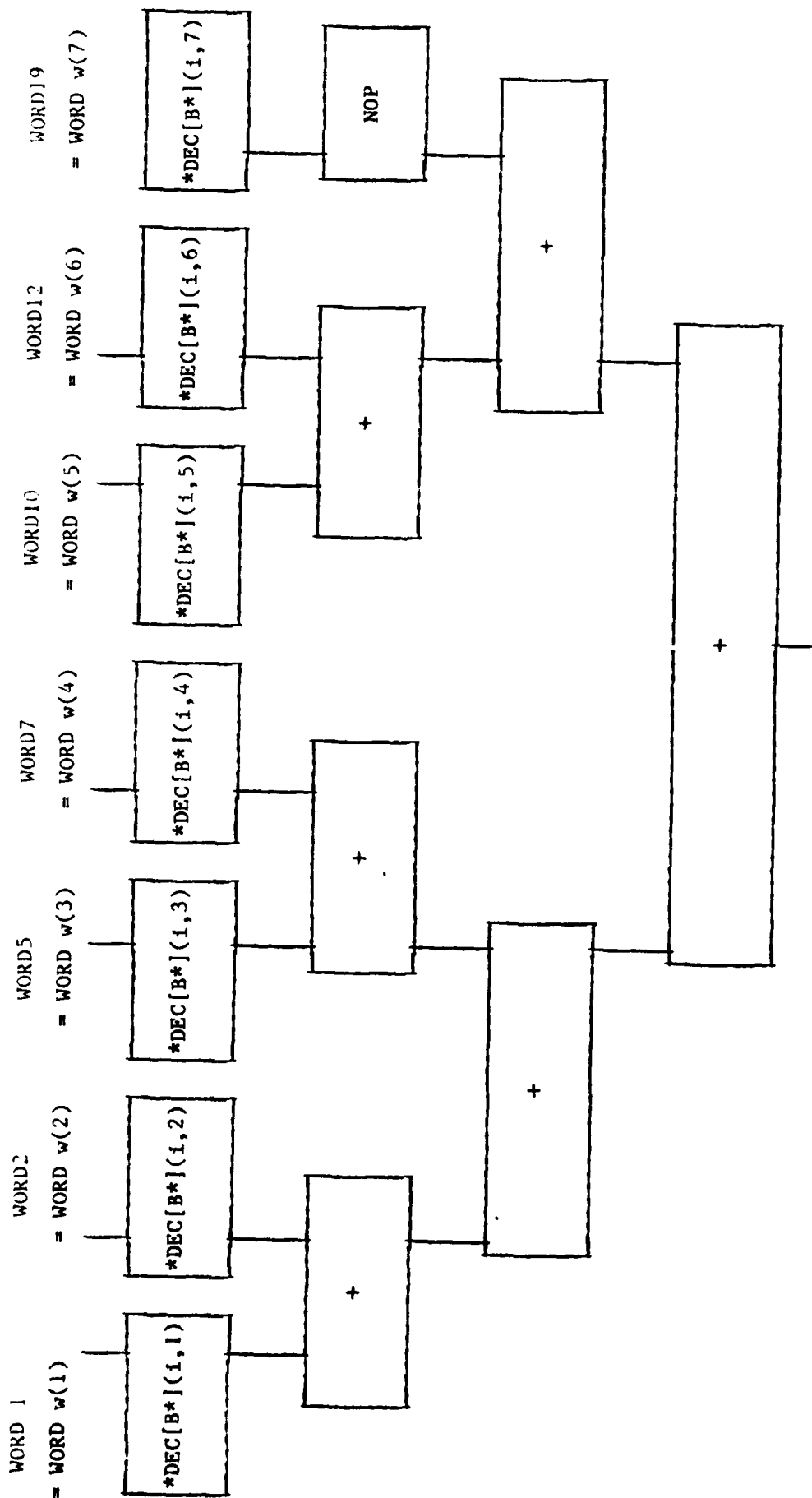


Figure 1.6.1

Example of decode for a
7-out-of-25 Bloom p/s/r.

Encode is similar.

Information flows downward.

NOP = no operation

$+$ = XOR of words, i.e. bitwise modulo 2 addition

On the face of things it would appear that one would have to use 5 cycles to fill in the (variable) 5-bit multiplicand WORD $w(j)$ into the box which multiplies by the (fixed) 5-bit multiplier $DEC[B^*](i,j)$, then take more than 5 additional cycles to perform the GF(32) multiplication, then 3 more cycles to move through the adders (the add operation is XOR). This would involve an output stream slower than the one bit per cycle input stream. This, however, is not the case. We will show below how to produce a one bit per cycle output stream, using appropriate hardware. Of course the output will lag the input in phase. In the case above the lag will be about 18 cycles.

Again we note the need to keep parallel channels synchronized. This means that even the plaintext channels will be "encoded" (or "decoded"). This will be done by multiplying by 1, then adding 0, then adding another 0, and so on for the proper number of steps.

1.7 Making hyperfast Bloom p/s/r processes. Stage 3. Hot precomputation.

Recall that we are considering the mean cases of parameter settings. Turning now from Stage 4, real-time on-line decode, to Stage 3, hot precomputation, we come to an important problem. You want to shorten the hot precomputation because you must store or lose received bits while it takes place. It turns out that the hot precomputation should be done somewhat differently for different parameter settings in the mean cases of parameter settings.

In Section 2.5 below we take up this matter in more detail. If k or $n-k$ is small, the hot precomputation proceeds quickly.

In summary, the only rub anywhere in the system occurs in the hot precomputation. And it is worst when k is close to $n/2$. In many applications, such as digital voice, where loss of one second's worth of transmission is tolerable, the rub can be ignored. In other applications, its presence may necessitate enough buffer memory to store several second's worth of received material.

Of course there is inevitably one other place where simple common sense dictates that expense is inevitable, not for memory to store signals but for memory and processing capability to do computations. In 16 bit applications in which $40,000 \leq k \leq n$ there are a lot of received channels and some very big (40,000 by 40,000) matrices to build. It is important to keep in mind the admonition that most systems with more than 256 channels are impractical. We return to this matter below.

1.8. Interfacing error control devices and cryptographic devices with p/s/r processes

p/s/r processes work best on channels which are virtually error-free while operative (like some optical fibers), but which can be rendered inoperative for long periods (e.g. by breaking the fibers). If the operative channels are also subject to intermittent errors then one should combine ordinary error control with p/s/r processes in the manner shown in Figure 1.8.1. First p/s/r encode, then error control encode, then transmit, then receive, then error control decode, then p/s/r decode. Doing an error control encode before the p/s/r encode would be silly. We will not belabor this point further.

Cryptographic encode should probably be placed before p/s/r encode and cryptographic decode after p/s/r decode, as in Figure 1.8.2. But this is a matter which will no doubt be determined by an appropriate branch of DOD, and we will therefore not treat it further.

Figure 1.8.3 shows the concatenation scheme for all three processes. All figures are to be understood as showing information flowing downward.

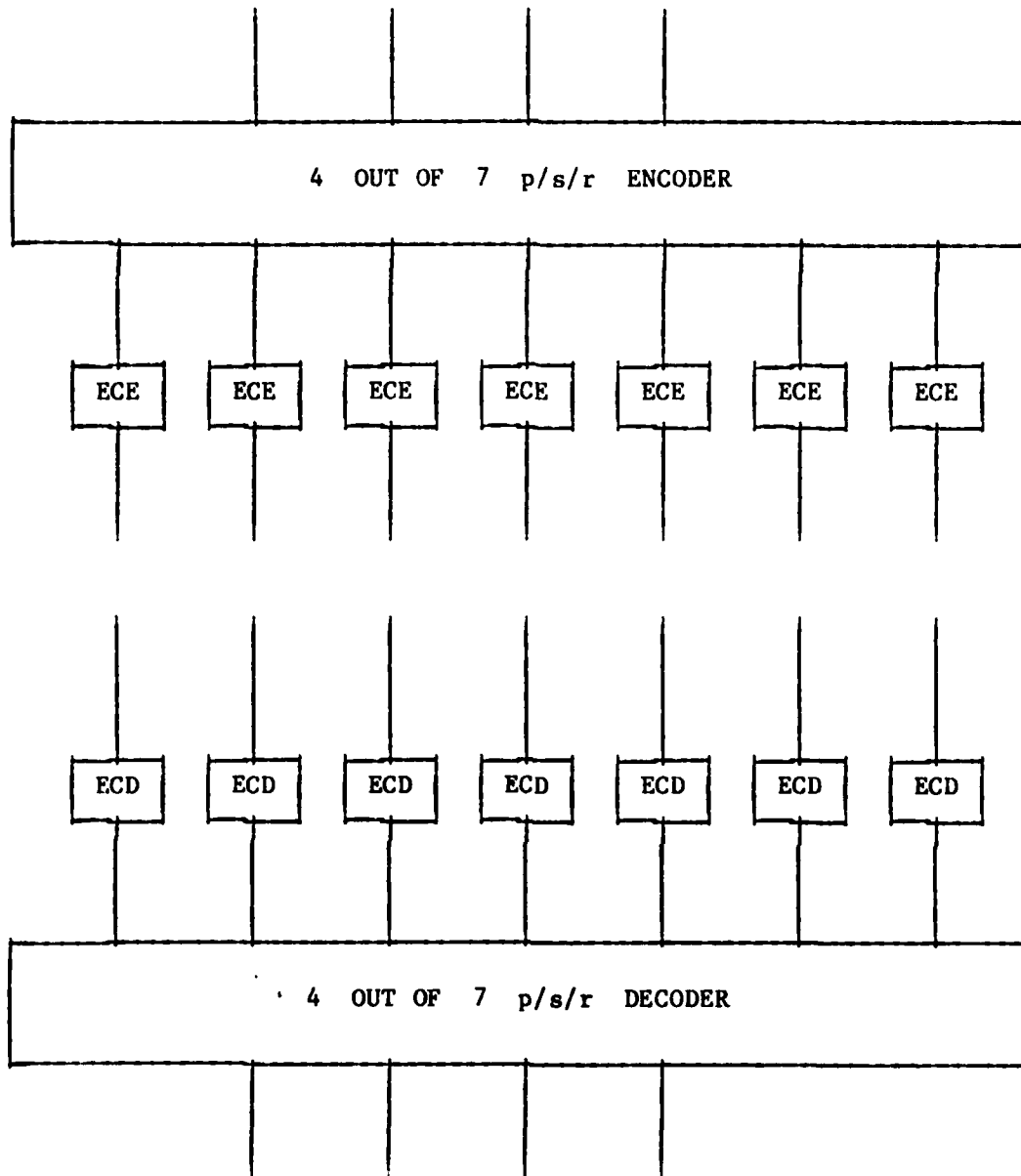


Figure 1.8.1

ECE = error control encode

ECD = error control decode

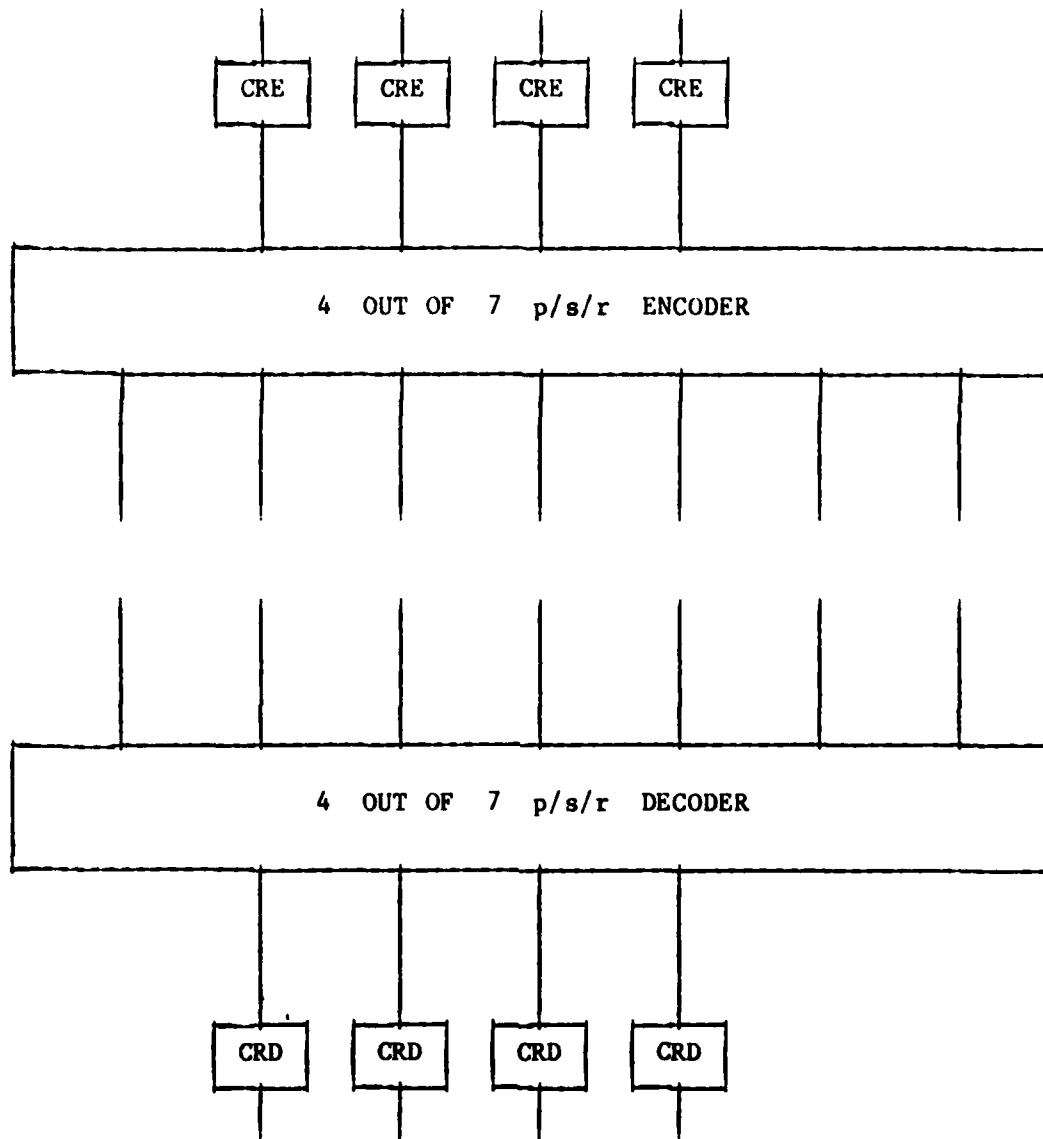


Figure 1.8.2

CRE = cryptographic encode

CRD = cryptographic decode

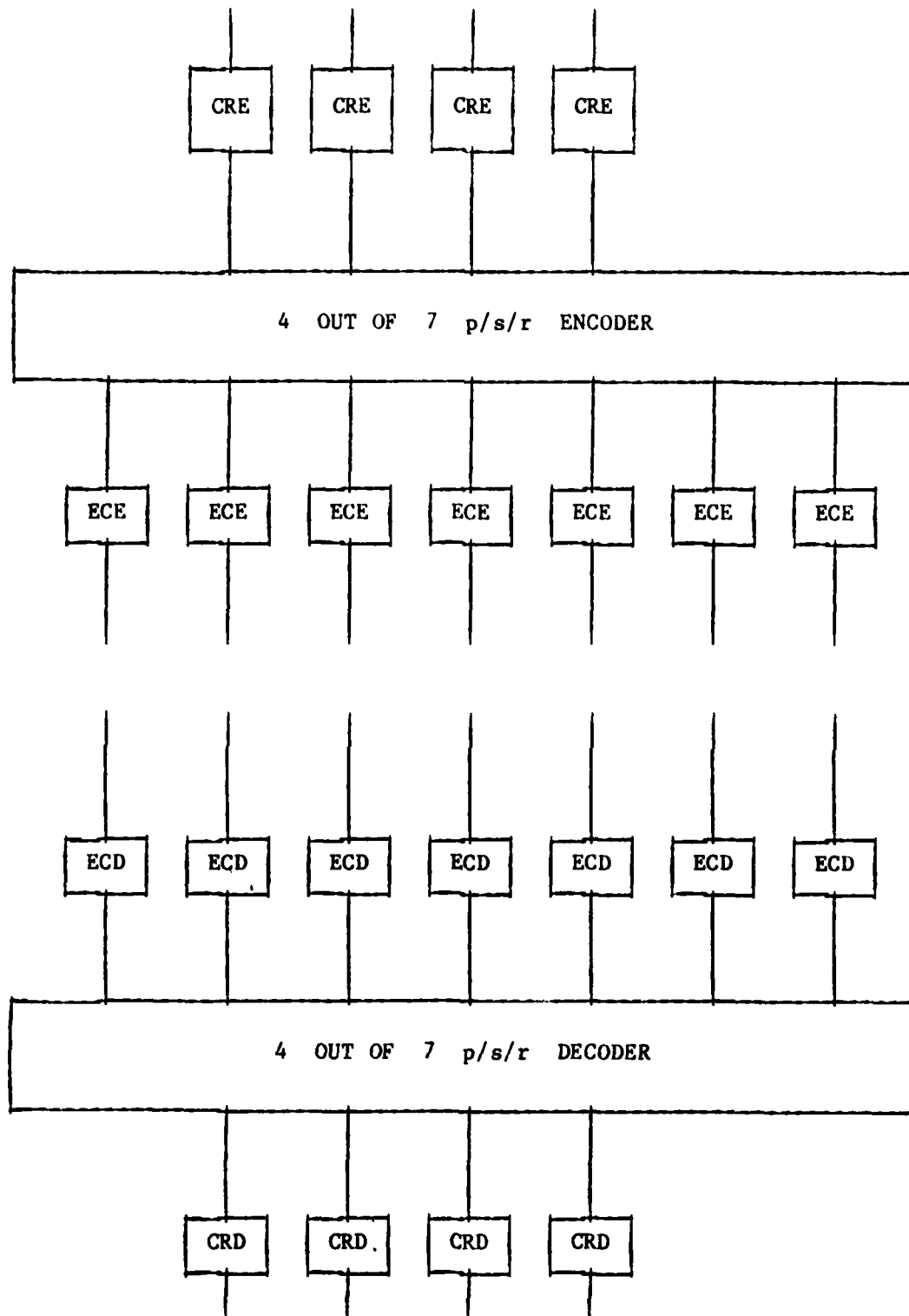


Figure 1.8.3

2. Detailed Narrative

2.1. Finite field arithmetic. Octal notation for polynomials and residue classes of polynomials.

It is no longer possible or desirable to avoid technicalities. We first make explicit the finite field arithmetic behind the Bloom p/s/r processes. $GF(2) = \mathbb{Z}/2\mathbb{Z}$ is the field with two elements. Its arithmetic (i.e. its add, +, subtract, -, multiply, *, and divide, /) is summarized in the tables

+	0	1	-	0	1	*	0	1	/	0	1
0	0	1	0	0	1	0	0	0	0	undefined	0
1	1	0	1	1	0	1	0	1	1	undefined	1

Thus $x + y = x - y$ for every $x, y \in GF(2)$, the only nonzero product is $1 * 1 = 1$, and division by zero is impossible (undefined). You can put these things another way. +, -, and * are modulo 2 operations, and you cannot divide by zero. Alternatively, + and - are XOR of bits (exclusive or), * is AND of bits, and you cannot divide by zero.

Let $p(x)$ be a polynomial over $GF(2)$ which is irreducible (unfactorable) over $GF(2)$. Examples of polynomials over $GF(2)$ which are irreducible over $GF(2)$ are:

x
 $x + 1$
 $x^2 + x + 1$
 $x^3 + x + 1$
 $x^4 + x + 1$
 $x^5 + x^2 + 1$
 $x^6 + x + 1$
 $x^7 + x^3 + 1$
 $x^8 + x^4 + x^3 + x^2 + 1$
 $x^{12} + x^6 + x^4 + x + 1$
 $x^{16} + x^{12} + x^3 + x + 1$

Examples of polynomials over $GF(2)$ which are reducible over $GF(2)$ (i.e. polynomials which can be factored) are:

$$\begin{aligned}x^2 &= x * x \\x^2 + x &= x * (x + 1) \\x^2 + 1 &= (x + 1) * (x + 1) \\x^4 + x^2 + 1 &= (x^2 + x + 1) * (x^2 + x + 1) \\x^4 + x^2 + x + 1 &= (x + 1) * (x^3 + x^2 + 1)\end{aligned}$$

Let n be a positive integer. The field $GF(2^n)$ is defined as follows. Let $p(x)$ be an n th degree (monic) polynomial over $GF(2)$ which is irreducible over $GF(2)$. Let $(p(x))$ be the principal ideal generated by $p(x)$ in the ring POL of polynomials over $GF(2)$. Then $GF(2^n)$ is the quotient

$$GF(2^n) = POL/(p(x)).$$

of the ring POL modulo the principal ideal generated by $p(x)$. For example if $p(x) = x^3 + x + 1$ then the version of $GF(8) = GF(2^3)$ gotten by setting

$$GF(8) = POL/(p(x)) = POL/(x^3 + x + 1)$$

consists of 8 residue classes modulo $p(x) = x^3 + x + 1$, namely

$$\begin{aligned}\underline{0} &= \langle 0, 0, 0 \rangle = \text{CLASS } (0) = \{0, x^3 + x + 1, \dots\} \\ \underline{1} &= \langle 0, 0, 1 \rangle = \text{CLASS } (1) = \{1, x^3 + x, \dots\} \\ \underline{2} &= \langle 0, 1, 0 \rangle = \text{CLASS } (x) = \{x, x^3 + 1, \dots\} \\ \underline{3} &= \langle 0, 1, 1 \rangle = \text{CLASS } (x+1) = \{x+1, x^3, \dots\} \\ \underline{4} &= \langle 1, 0, 0 \rangle = \text{CLASS } (x^2) = \{x^2, x^3 + x^2 + x + 1, \dots\} \\ \underline{5} &= \langle 1, 0, 1 \rangle = \text{CLASS } (x^2 + 1) = \{x^2 + 1, x^3 + x, \dots\} \\ \underline{6} &= \langle 1, 1, 0 \rangle = \text{CLASS } (x^2 + x) = \{x^2 + x, x^3 + x^2 + 1, \dots\} \\ \underline{7} &= \langle 1, 1, 1 \rangle = \text{CLASS } (x^2 + x + 1) = \{x^2 + x + 1, x^3, \dots\}\end{aligned}$$

It is too tedious to use a notation such as

$$\text{CLASS } (x^2 + x)$$
or
$$\langle 1, 0, 1 \rangle$$
or
$$\{x^2 + x, x^3 + x^2 + 1, x^4, \dots\}$$

for a member of $\text{GF}(8)$. Therefore we adopt the octal notation used in the MIT Press book of Peterson and Weldon on error correcting codes. An arabic numeral with neither overbar nor underbar is a whole number. Thus

$$7 = \text{VII} = \text{seven},$$

the number of days in the week. An arabic numeral with an overbar is a polynomial over $\text{GF}(2)$. Thus

$$\overline{7} = \langle 1, 1, 1 \rangle = x^2 + x + 1.$$

And an arabic numeral with an underbar is a residue class (modulo some agreed upon irreducible polynomial $p(x)$) to which a polynomial $q(x)$ belongs. Thus if $p(x) = x^3 + x + 1$ is agreed upon in advance then

$$\begin{aligned}
 \underline{7} &= \{x^2 + x + 1, x^3 + x^2, x^4 + 1, x^5 + x^3 + x + 1, \dots\} \\
 &= \{\overline{7}, \overline{14}, \overline{21}, \overline{53}, \dots\} \\
 &= \text{CLASS } (\overline{7}) \bmod (\overline{13})
 \end{aligned}$$

is the residue class modulo $x^3 + x + 1$ whose lowest degree member is $\overline{7} = x^2 + x + 1$.

We now agree on polynomials over $\text{GF}(2)$ of degrees 2, 3, 4, 5, 6, 7, 8, 12 and 16. Each of them is irreducible over $\text{GF}(2)$. In fact, each of them is a primitive irreducible polynomial over $\text{GF}(2)$. There is no need to describe the notion of primitive here. Suffice it to say that it is a convenience, and is explained in Peterson and Weldon.

There are nine standard polynomials to be understood everywhere below. They are the polynomials on which our version of $\text{GF}(4)$, $\text{GF}(8)$, $\text{GF}(16)$, $\text{GF}(32)$, $\text{GF}(64)$, $\text{GF}(128)$, $\text{GF}(256)$, $\text{GF}(4,096)$ and $\text{GF}(65,536)$ are based. It is, of course, well known that there is (up to isomorphism) only one Galois field of any given size.

The nine standard polynomials are

$$\overline{7} = x^2 + x + 1$$

$$\overline{13} = x^3 + x + 1$$

$$\overline{23} = x^4 + x + 1$$

$$\overline{45} = x^5 + x^2 + 1$$

$$\overline{103} = x^6 + x + 1$$

$$\overline{211} = x^7 + x^3 + 1$$

$$\overline{435} = x^8 + x^4 + x^3 + x^2 + 1$$

$$\overline{10123} = x^{12} + x^6 + x^4 + x + 1$$

$$\overline{210013} = x^{16} + x^{12} + x^3 + x + 1$$

Members of $GF(2^4) = GF(16)$ can thus be represented as 4-bit words, i.e. "numbers" expressible by two (underbarred) octal arabic numerals, neither of which is 8 or 9. Members of $GF(2^8) = GF(256)$ "are" 8 bit words, i.e. "numbers" expressible by three (underbarred) octal arabic numerals (8 and 9 will not be used). For $GF(2^{12}) = GF(4,096)$ we use 12 bit words, i.e. foursomes of underbarred arabic octal numerals (no 8 or 9 allowed). For $GF(2^{16}) = GF(65,536)$ we use 16 bit words, underbarred 6 "digit" arabic numerals (with no occurrence of 8 or 9).

To exemplify the arithmetic of $GF(2^n)$ we will give tables for:

$$GF(4) \text{ as } POL/(x^2 + x + 1) = POL/(\overline{7})$$

$$GF(8) \text{ as } POL/(x^3 + x + 1) = POL/(\overline{13})$$

$$GF(16) \text{ as } POL/(x^4 + x + 1) = POL/(\overline{23})$$

They are contained in Appendix B.

2.2 The linear algebra of Bloom p/s/r processes.

As noted above, the extreme parameter setting cases

$$(k,n) = (0,n)$$

$$(k,n) = (1,n)$$

$$(k,n) = (n,n)$$

require no coding. The extreme parameter setting case

$$(k,n) = (n-1,n)$$

can be very simply coded and decoded using only $GF(2)$, and without cold, cool or hot precomputation. Thus we will consider the mean parameter setting cases, i.e. the cases involving (k,n) such that

$$2 \leq k \leq n-2 \leq n \leq 2^{\dagger b} = Q \leq 65,536$$

Here b is a parameter describing the size of (i.e. number of bits in) the computer word to be used in practical implementations. Its place in the scheme of things will be obvious below.

Let us begin with the $2^{\dagger b}$ by $2^{\dagger b}$ (i.e. Q by Q) Vandermonde matrix with entries in $GF(2^{\dagger b}) = GF(Q)$. This square matrix VAN is of the form

$$VAN = \begin{bmatrix} \underline{1} & \underline{0} & \underline{0} & \underline{0} & \dots & \underline{0} & \underline{0} \\ \underline{1} & \underline{1} & \underline{1} & \underline{1} & \dots & \underline{1} & \underline{1} \\ \underline{1} & \underline{2} & \underline{2^{\dagger 2}} & \underline{2^{\dagger 3}} & \dots & \underline{2^{\dagger(b-2)}} & \underline{2^{\dagger(b-1)}} \\ \underline{1} & \underline{2^{\dagger 2}} & \underline{2^{\dagger 4}} & \underline{2^{\dagger 6}} & \dots & \underline{2^{\dagger 2(b-2)}} & \underline{2^{\dagger 2(b-1)}} \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ \underline{1} & \underline{2^{\dagger(b-3)}} & \underline{2^{\dagger 2(b-3)}} & \underline{2^{\dagger 3(b-3)}} & \dots & \underline{2^{\dagger(b-3)(b-2)}} & \underline{2^{\dagger(b-3)(b-1)}} \\ \underline{1} & \underline{2^{\dagger(b-2)}} & \underline{2^{\dagger 2(b-2)}} & \underline{2^{\dagger 3(b-2)}} & \dots & \underline{2^{\dagger(b-2)(b-2)}} & \underline{2^{\dagger(b-2)(b-1)}} \end{bmatrix}$$

Note that the bases are (underbarred) members of $GF(Q)$ and the exponents are (unbarred) integers. It is a fact that $\underline{2}$ is a primitive element in $GF(2^{\dagger b})$ if $GF(2^{\dagger b})$ is realized as $POL/(p(x))$ where

$p(x)$ is a primitive polynomial. We will always use fields of this form. Examples of Vandermonde matrices are

$$\text{VAN} = \begin{bmatrix} \underline{1} & \underline{0} & \underline{0} & \underline{0} \\ \underline{1} & \underline{1} & \underline{1} & \underline{1} \\ \underline{1} & \underline{2} & \underline{3} & \underline{1} \\ \underline{1} & \underline{3} & \underline{2} & \underline{1} \end{bmatrix}$$

in $\text{GF}(4) = \text{POL}/(x^2 + x + 1) = \text{POL}/(\underline{7})$, and

$$\text{VAN} = \begin{bmatrix} \underline{1} & \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{0} \\ \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} \\ \underline{1} & \underline{2} & \underline{4} & \underline{3} & \underline{6} & \underline{7} & \underline{5} & \underline{1} \\ \underline{1} & \underline{4} & \underline{6} & \underline{5} & \underline{2} & \underline{3} & \underline{7} & \underline{1} \\ \underline{1} & \underline{3} & \underline{5} & \underline{4} & \underline{7} & \underline{2} & \underline{6} & \underline{1} \\ \underline{1} & \underline{6} & \underline{2} & \underline{7} & \underline{4} & \underline{5} & \underline{3} & \underline{1} \\ \underline{1} & \underline{7} & \underline{3} & \underline{2} & \underline{5} & \underline{6} & \underline{4} & \underline{1} \\ \underline{1} & \underline{5} & \underline{7} & \underline{6} & \underline{3} & \underline{4} & \underline{2} & \underline{1} \end{bmatrix}$$

in $\text{GF}(8) = \text{POL}/(x^3 + x + 1) = \text{POL}/(\underline{13})$. See Appendix C for examples of Vandermonde matrices, for various fields.

Now let $\text{LEF}[k]$ be a special Q by k submatrix of VAN . It consists of the first k columns of VAN . In our $\text{GF}(8)$ example

$$\text{LEF}[3] = \begin{bmatrix} \underline{1} & \underline{0} & \underline{0} \\ \underline{1} & \underline{1} & \underline{1} \\ \underline{1} & \underline{2} & \underline{4} \\ \underline{1} & \underline{4} & \underline{6} \\ \underline{1} & \underline{3} & \underline{5} \\ \underline{1} & \underline{6} & \underline{2} \\ \underline{1} & \underline{7} & \underline{3} \\ \underline{1} & \underline{5} & \underline{7} \end{bmatrix}$$

It is a well known property of Vandermonde matrices that every k by k submatrix of $\text{LEF}[k]$ is nonsingular whenever k satisfies the inequalities

$$2 \leq k \leq Q = 2^{\dagger}b .$$

Thus the rows of $\text{LEF}[k]$ can be regarded as a collection of $2^{\dagger}b = Q$ vectors in general position in a k dimensional vector space V over $\text{GF}(2^{\dagger}b) = \text{GF}(Q)$. But recall that $2 \leq k \leq n-2 \leq n \leq Q$. This means that we have the wherewithal to build a Bloom k -out-of- n p/s/r process. Consider any list $w(1), w(2), \dots, w(k)$ of distinct row indices of $\text{LEF}[k]$, i.e. any injection

$$w: \{1, 2, \dots, k\} \rightarrow \{1, 2, \dots, Q\}.$$

There is obviously a Q by k (coding) matrix COD_w corresponding to this w such that

$$\text{ROW}[i] = \sum \text{COD}_w(i,j) \text{ROW}[w(j)]$$

for every positive integer $i \leq Q$. In particular

$$\begin{aligned} \text{ROW}[i] &= \sum \text{COD}_e(i,j) \text{ROW}[e(j)] \\ &= \sum \text{COD}_e(i,j) \text{ROW}[j] \end{aligned}$$

when e is the identity injection. All three sums above are over positive integer $j \leq k$.

The Bloom k -out-of- n p/s/r process now works as follows. Let $I(1), I(2), \dots, I(k)$ be the b -bit plaintext words the sender S has on source channels $1, 2, \dots, k$ at time t . The sender encodes them to form b -bit words $H(1), H(2), \dots, H(n)$ for sending along broadcast channels $1, 2, \dots, n$ as follows

$$H(i) = \sum \text{COD}_e(i, j) I(j)$$

for positive integer $i \leq n$, where the sum is over positive integer $j \leq k$. When the receiver R ascertains that channels $w(1), w(2), \dots, w(k)$ are operative, he decodes by finding

$$I(i) = \sum \text{COD}_w(i, j) H(w(j))$$

for positive integer $i \leq k$, where the sum is over positive integer $j \leq k$.

Before looking at implementation in the four stages we make a few comments. First, encoding is a process which depends only on k and Q , not on n (except in the trivial sense that you don't bother to encode any messages $H(i)$ for channels $n+1, n+2, \dots, Q$) and not on which channels are operative and which are inoperative. After all, the sender is not likely to know which channels are operative. Mathematically speaking, encoding makes use only of the (fixed) identity injection e .

Decoding, on the other hand, makes use of the (variable) injection w which embodies information known only to the receiver, namely which channels $w(1), w(2), \dots, w(k)$ are operative. So decoding depends on k, w and Q . Consequently decoding depends implicitly on n , since $1 \leq w(i) \leq n$ for every positive integer $i \leq k$.

If either sender S or receiver R can profit by taking n into account in a more explicit fashion in their calculations, they are free to do so. But they don't have to. We will show below how to take advantage of a knowledge of n .

Comparing this description with the YLYK Ltd. proposal, the reader will note our assumption that

$$n \leq 2tb = Q.$$

That proposal held forth the possibility of the inequality

$$n \leq Q + 2$$

in many cases.

We abandoned this tack, fine tuning the field size, for four reasons:

1. It shortens word size by only one or two bits where it is possible;
2. It complicates coding and decoding where it is possible;
3. It is a very difficult problem to determine all the cases in which it is possible. See the MacWilliams and Sloane book on error correcting codes for more on this;
4. We now know how to achieve the desired goal of attaining hyperfast Bloom p/s/r processes without fine tuning the field size. The hyperfast real-time on-line decode is attained in a different way, by use of systolic multipliers, as we shall see below. Moreover, fine tuning field size is of no appreciable utility in attacking the other crucial problem, shortening the duration of Stage 3, the hot precomputation.

2.3. The first stage of computation in the mean cases of the Bloom p/s/r process, the manufacturer's cold precomputation

Recall that we have a field

$$GF(2^b) = GF(Q)$$

and that

$$2 \leq k \leq n-2 \leq n \leq 2^b = Q.$$

The entire problem of encoding and decoding in a k-out-of-n Bloom p/s/r process amounts to this. For each injection

$$w: \{1, 2, \dots, k\} \rightarrow \{1, 2, \dots, n\}$$

(including the identity injection $w = e$) find the k by k matrix COD_w such that

$$\text{ROW}[i] = \sum \text{COD}_w(i, j) \text{ROW}[w(j)]$$

where the sum is over positive integer $j \leq k$, and where $\text{ROW}[i]$ is the i th row of the Q by k matrix $\text{LEF}[k]$. Recall that $\text{LEF}[k]$ consists of the first k columns of the Q by Q Vandermonde matrix VAN over $\text{GF}(Q)$. Once COD_e is found, form

$$H(i) = \sum \text{COD}_e(i, j) I(j)$$

(where the sum is over every positive integer $j \leq k$) for every positive integer $i \leq n$ to encode. Once w is chosen and COD_w is found, form

$$I(i) = \sum \text{COD}_w(i, j) H(w(j))$$

(where the sum is over every positive integer $j \leq k$) for every positive integer $i \leq k$ to decode.

Obviously it is desirable to carry out computations as early as possible. We have agree to send the k plaintext messages $I(1), I(2), \dots, I(k)$ (i.e. members of $\text{GF}(2^b) = \text{GF}(Q)$, i.e. b -bit words) down channels $1, 2, \dots, k$ respectively. These words are unaltered. They are transmitted as is.

$$H(1) = I(1)$$

$$H(2) = I(2)$$

$$\vdots$$

$$\vdots$$

$$\vdots$$

$$H(k) = I(k)$$

What we need is the encoding for channels $k+1, k+2, \dots, n$. In other words we need to express rows $k+1, k+2, \dots, k+n$ of $\text{LEF}[k]$ in terms of rows $1, 2, \dots, k$. To say we need dependences is to say we need vanishing linear combinations of the rows of $\text{LEF}[k]$. We need, therefore, a basis for the left kernel of $\text{LEF}[k]$ (The left kernel of a Q by k matrix L is the set of all length Q row vectors r such that rQ is the length k row vector with all zero entries). Let us take $\text{GF}(8)$ as an example.

$$\text{LEF}[8] = \text{VAN} = \begin{bmatrix} \underline{1} & \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{0} \\ \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} \\ \underline{1} & \underline{2} & \underline{4} & \underline{3} & \underline{6} & \underline{7} & \underline{5} & \underline{1} \\ \underline{1} & \underline{4} & \underline{6} & \underline{5} & \underline{2} & \underline{3} & \underline{7} & \underline{1} \\ \underline{1} & \underline{3} & \underline{5} & \underline{4} & \underline{7} & \underline{2} & \underline{6} & \underline{1} \\ \underline{1} & \underline{6} & \underline{2} & \underline{7} & \underline{4} & \underline{5} & \underline{3} & \underline{1} \\ \underline{1} & \underline{7} & \underline{3} & \underline{2} & \underline{5} & \underline{6} & \underline{4} & \underline{1} \\ \underline{1} & \underline{5} & \underline{7} & \underline{6} & \underline{3} & \underline{4} & \underline{2} & \underline{1} \end{bmatrix}$$

is nonsingular.

$$\text{LEF}[7] = \begin{bmatrix} \underline{1} & \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{0} \\ \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} \\ \underline{1} & \underline{2} & \underline{4} & \underline{3} & \underline{6} & \underline{7} & \underline{5} \\ \underline{1} & \underline{4} & \underline{6} & \underline{5} & \underline{2} & \underline{3} & \underline{7} \\ \underline{1} & \underline{3} & \underline{5} & \underline{4} & \underline{7} & \underline{2} & \underline{6} \\ \underline{1} & \underline{6} & \underline{2} & \underline{7} & \underline{4} & \underline{5} & \underline{3} \\ \underline{1} & \underline{7} & \underline{3} & \underline{2} & \underline{5} & \underline{6} & \underline{4} \\ \underline{1} & \underline{5} & \underline{7} & \underline{6} & \underline{3} & \underline{4} & \underline{2} \end{bmatrix}$$

has rank 7. Therefore its kernel has dimension 1 and a calculation shows that it is spanned by the row vector

$$[\underline{1} \ \underline{1} \ \underline{1} \ \underline{1} \ \underline{1} \ \underline{1} \ \underline{1} \ \underline{1}]$$

$$\text{LEF}[6] = \begin{bmatrix} \underline{1} & \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{0} \\ \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} \\ \underline{1} & \underline{2} & \underline{4} & \underline{3} & \underline{6} & \underline{7} \\ \underline{1} & \underline{4} & \underline{6} & \underline{5} & \underline{2} & \underline{3} \\ \underline{1} & \underline{3} & \underline{5} & \underline{4} & \underline{7} & \underline{2} \\ \underline{1} & \underline{6} & \underline{2} & \underline{7} & \underline{4} & \underline{5} \\ \underline{1} & \underline{7} & \underline{3} & \underline{2} & \underline{5} & \underline{6} \\ \underline{1} & \underline{5} & \underline{7} & \underline{6} & \underline{3} & \underline{4} \end{bmatrix}$$

has rank six. So its kernel contains the kernel of LEF[7]. A calculation shows that it is spanned by the row vectors

$$[\underline{1} \ \underline{1} \ \underline{1} \ \underline{1} \ \underline{1} \ \underline{1} \ \underline{1} \ \underline{1}]$$

and

$$[\underline{7} \ \underline{6} \ \underline{2} \ \underline{5} \ \underline{3} \ \underline{4} \ \underline{1} \ \underline{0}]$$

Similarly, one easily verifies that

$$\begin{bmatrix} \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} \\ \underline{7} & \underline{6} & \underline{2} & \underline{5} & \underline{3} & \underline{4} & \underline{1} & \underline{0} \\ \underline{2} & \underline{3} & \underline{2} & \underline{1} & \underline{3} & \underline{1} & \underline{0} & \underline{0} \end{bmatrix} \begin{bmatrix} \underline{1} & \underline{0} & \underline{0} & \underline{0} & \underline{0} \\ \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} \\ \underline{1} & \underline{2} & \underline{4} & \underline{3} & \underline{6} \\ \underline{1} & \underline{4} & \underline{6} & \underline{5} & \underline{2} \\ \underline{1} & \underline{3} & \underline{5} & \underline{4} & \underline{7} \\ \underline{1} & \underline{6} & \underline{2} & \underline{7} & \underline{4} \\ \underline{1} & \underline{7} & \underline{3} & \underline{2} & \underline{5} \\ \underline{1} & \underline{5} & \underline{7} & \underline{6} & \underline{3} \end{bmatrix} = \begin{bmatrix} \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{0} \\ \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{0} \\ \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{0} \end{bmatrix}$$

Going on in this way we arrive at the encode normal form ENF matrix over GF(8):

$$\begin{bmatrix} \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{1} \\ \underline{7} & \underline{2} & \underline{6} & \underline{5} & \underline{3} & \underline{4} & \underline{1} & \underline{0} \\ \underline{2} & \underline{3} & \underline{2} & \underline{1} & \underline{3} & \underline{1} & \underline{0} & \underline{0} \\ \underline{3} & \underline{5} & \underline{2} & \underline{5} & \underline{1} & \underline{0} & \underline{0} & \underline{0} \\ \underline{4} & \underline{3} & \underline{6} & \underline{1} & \underline{0} & \underline{0} & \underline{0} & \underline{0} \\ \underline{3} & \underline{2} & \underline{1} & \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{0} \end{bmatrix} .$$

This matrix ENF is a $Q-2$ by Q matrix with 1 in the $(j, Q-j+1)$ th entry and with 0 entries everywhere below these "antidiagonal" 1s. In fact the matrix product $ENF * VAN$ is a $Q-2$ by Q matrix with zeros above the "antidiagonal". Given any Q by Q Vandermonde matrix for $GF(Q)$ it is elementary linear algebra to find the $Q-2$ by Q matrix ENF such that:

1. For each positive integer $j \leq Q-2$ the top j rows of ENF form a basis for the left kernel of $LEF[Q-j]$
2. The antidiagonal entries (i.e. $ENF(j, Q-j+1)$ for every positive integer $j \leq Q-2$) are 1
3. The entries below the antidiagonal are 0.

This is the substance of the manufacturer's cold precomputation, Stage 1. A computer program incorporating this precomputation is contained in Appendix H. It could take months on an IBM 370 and still be perfectly satisfactory, since it will be done just once before the devices are fabricated. In fact the $GF(16)$ computation takes seconds on an IBM PC. The $GF(16)$ ENF is a 14 by 16 matrix whose entries are 4-bit words. See Appendix D for examples of ENF matrices for various fields.

The $GF(256)$ cold precomputation, even without the shortcuts employed in Appendix H, takes far fewer than a billion machine cycles, i.e. a few minutes of mainframe time. To store its output requires $254*256 = 65,024$ bytes of ROM. The $GF(4,096)$ and $GF(65,536)$ cold precomputations take longer.

Since finding a kernel basis and triangularizing its matrix takes a small constant times the cube of the dimension of the vector space, finding a 4094 by 4096 ENF matrix for $GF(4,096 = GF(2^{12}))$ could take as many 10^{13} single precision integer operations and single word logical operations on an IBM 370. This could take months. To store the output you would need more than 200 megabits of ROM.

To find a 65,534 by 65,536 ENF matrix over $GF(65,536) = GF(2^{16})$ is a bigger task. Here we are talking about a fair sized multiple of 2^{48} operations, say 10^{17} to be on the safe side. Of course, this assumes no parallelism in the computer. But parallelism and vector structure are keynotes of the computation. However it looks like months of calculation on better adapted machines such as a CRAY I or the new MPP being put up at NASA, both of them well-suited to the sort of linear algebra computations required. It also means scrapping the PASCAL program in Appendix A and writing code which exploits the peculiarities of the machine it runs on. Also, storing its output is nontrivial. This output consists of $65,534 * 65,536 = 4,294,836,224$ 16-bit words. This means almost 9 gigabytes of ROM in the devices which implement such p/s/r processes.

What about larger fields? It seems doubtful that they can be exploited economically in the 1980s, or that they would be used even if computations were cheap. Some objections are:

- i. 65,537 channels is a lot of channels. Is there a plausible application of k out of n p/s/r processes in a situation where $n > 2^{16} = 65,536$?
- ii. Fields larger than $GF(2^{16})$ cannot be handled on a 16 bit microprocessor without adopting unnatural expedients which slow things down.
- iii. Stage 1, the cold precomputation stage in which ENF is formed, gets expensive and time consuming in $GF(2^b)$ for $b > 16$. For example production of an ENF for $GF(2^{20})$ looks like a multiple of 2^{60} operations on a Von Neumann machine, say 10^{20} operations.
- iv. Storing the ENF in fields bigger than $GF(65,536)$ requires more than 9 gigabytes of ROM.

Summarizing the first stage, the manufacturer's cold precomputation stage, we see that the $2^{\dagger b} - 2$ by $2^{\dagger b}$ encode normal form matrix ENF has the following properties (pessimistic estimates):

Galois field	time to produce ENF	space to store ENF
$GF(16) = GF(2^{\dagger 4})$	PC minutes	1 k bits
$GF(256) = GF(2^{\dagger 8})$	mainframe minutes	600 k bits
$GF(4,096) = GF(2^{\dagger 12})$	mainframe months	300 m bits
$GF(65,536) = GF(2^{\dagger 16})$	supercomputer years	70 g bits

2.4 The second stage of computation in the mean cases of the Bloom p/s/r process, the sender's cool precomputation.

Recall that we have, once for all, chosen

$$GF(2^{\dagger b}) = GF(Q)$$

Thus the sender S must take k b -bit words at time t and encode this information into n b -bit words for transmission. Moreover

$$2 \leq k \leq n-2 \leq n \leq 2^{\dagger b} = Q$$

The ENF matrix is available to both sender and receiver. It contains information about VAN or, more specifically, about $LEF[2], LEF[3], \dots, LEF[Q-1]$. The first row of ENF expresses the Q th row of $LEF[Q-1]$ (and therefore of $LEF[Q-2], \dots, LEF[2]$) as a linear combination of its first $Q-1$ rows. The second row of ENF expresses the $(Q-1)$ st row of $LEF[Q-2]$ (and therefore of $LEF[Q-3], \dots, LEF[2]$) as a linear combination of its first $Q-2$ rows. And so on, to the bottom row (the $(Q-2)$ nd row) of ENF. This row of ENF expresses the third row of $LEF[2]$ in terms of the first two rows of $LEF[2]$.

Once k and n are agreed upon, the sender S and receiver R fix their attention on $LEF[k]$. They can ignore its bottom $Q - n$ rows. Thus they are looking at the upper left n by k submatrix $UPLEF[n,k]$ of VAN. Clearly, the dependences they both need to know among the rows of $LEF[k]$ (or, equivalently, of $UPLEF[n,k]$) are all contained (implicitly at least) in rows $Q - n + 1, Q - n + 2, \dots, Q - k$ of ENF.

For example a 3-out-of-7 p/s/r over $GF(8)$ is based on knowing the dependences among the first seven rows of

$$LEF[3] = \begin{bmatrix} \underline{1} & \underline{0} & \underline{0} \\ \underline{1} & \underline{1} & \underline{1} \\ \underline{1} & \underline{2} & \underline{4} \\ \underline{1} & \underline{4} & \underline{6} \\ \underline{1} & \underline{3} & \underline{5} \\ \underline{1} & \underline{6} & \underline{2} \\ \underline{1} & \underline{7} & \underline{3} \\ \underline{1} & \underline{5} & \underline{7} \end{bmatrix}$$

and these are all expressed in rows

$$8 - 7 + 1 = 2$$

$$8 - 7 + 2 = 3$$

.

.

.

$$8 - 3 = 5$$

of ENF, i.e. in the matrix

$$\text{MID}[2,5] = \begin{bmatrix} \underline{7} & \underline{2} & \underline{6} & \underline{5} & \underline{3} & \underline{4} & \underline{1} & \underline{0} \\ \underline{2} & \underline{3} & \underline{2} & \underline{1} & \underline{3} & \underline{1} & \underline{0} & \underline{0} \\ \underline{3} & \underline{5} & \underline{2} & \underline{5} & \underline{1} & \underline{0} & \underline{0} & \underline{0} \\ \underline{4} & \underline{3} & \underline{6} & \underline{1} & \underline{0} & \underline{0} & \underline{0} & \underline{0} \end{bmatrix}$$

Let $\text{MID}[Q-n+1, Q-k]$ be the $n-k$ by Q submatrix of ENF consisting of rows $Q-n+1, Q-n+2, \dots, Q-k$ of ENF. We now have the only matrix of interest to the sender S and the receiver R during this communication session using this k -out-of- n p/s/r. The last $Q-n$ columns of $\text{MID}[Q-n+1, Q-k]$ are, of course, zero. So they can, and will, be ignored in implementations. But a theoretical discussion proceeds more smoothly if we speak of all of $\text{MID}[Q-n+1, Q-k]$. The sender S sends channels $1, 2, \dots, k$ in the clear (i.e. uncoded). But he needs to know how to encode channels $k+1, k+2, \dots, n$. To do this the sender S can use elementary row operations to go from the already "triangularized" $\text{MID}[Q-n+1, Q-k]$ to a "diagonalized" form SEN in which column $k+1$ has all zeros except for a 1 in the bottom row. Column $k+2$ has all zeros except for a 1 in the row above the bottom row, and column n has all zeros except for a 1 in the top row. This is a trivial variant of the process of reducing to Hermite normal form. Once he has produced the matrix $\text{SEN} = \text{SEN}[k, n, Q]$, the sender S has finished his cool precomputation and he can start to encode and send. His encode amounts to

$$H(j) = \sum \text{SEN}(j, g) I(g) = \sum \text{SEN}[k, n, Q](j, g) I(g)$$

for each $j \in \{k+1, k+2, \dots, n\}$, where the sum is over positive integers $g \leq k$.

The cool precomputation is linear algebraic, like the cold precomputation, but it is shorter. For a k -out-of- n p/s/r process it involves bringing an already triangularized matrix with $n-k$ rows to a diagonalized form. This involves about $(n-k)(n-k+1)/2$ row operations. Therefore, approximately $n(n-k)(n-k+1)$ arithmetical operations are involved, i.e. subtractions/additions (XORs) and Galois

field multiplication. It is only necessary to find $n-k$ Galois field reciprocals if you do things carefully. This is helpful, since Galois field divisions require the Euclidean algorithm and are much slower than Galois field multiplications (unless we merely store arithmetical tables, an attractive expedient if $Q \leq 256$).

Consider a device built with Q synchronized parallel processors and a stored multiplication table they can all draw the same product from simultaneously. On such a device it would take about $c(n-k)+2$ machine cycles for the computation, where c is around 10. Thus for $GF(16) = GF(Q)$ (i.e. $2 \leq k \leq n-2 \leq n \leq 16$) we need 16 4-bit processors, a 16 by 16 table of 4-bit words (1 k bit ROM) and around $10 * 14 + 2 = 1960$ machine cycles for parallel implementation on 16 processors. It would take about 50,000 cycles for implementation by one processor. This means a delay of several milliseconds before the sender S can send. For $GF(256)$ we need 256 8-bit processors, a 256 by 256 table of 8-bit words (512 k bit ROM) and a delay of the order $10 * 254 + 2 = 645,160$ machine cycles (i.e. about a second) before sending could start. With only one processor this delay could rise to $256 * 645,160$ which is approximately 200 million machine cycles. So it could take many seconds before transmission began. Of course the sender could send plaintext over the first k channels while waiting for the coding process for the last $n - k$ channels to be found.

If some important (k,n) pairs were incorporated into firmware the sender's cool precomputation could be made part of the manufacturer's cold precomputation.

Turning to $GF(65,536)$ a parallel implementation would need 65,536 16-bit processors, and $16 * 65,536 + 2$ bits of ROM (i.e. 70 gigabits) The delay before sending could be as much as 40 billion machine cycles, an hour or so. Using just one 16-bit processor and doing the multiplications on the fly to dispense with the need for ROM could raise the delay before sending to years.

So, yet again, we see indications that 65,000 channels is a lot of channels to spread your messages among. But 256 channels once again looks very promising.

Let us summarize the second stage, the sender's cool precomputation stage. He extracts (from ENF) and row reduces (to a sort of Hermite normal form) the matrix $MID[Q-n+1, Q-k]$ to produce an $n - k$ by n matrix $SEN[k, n, Q]$. This matrix describes how to form the encoded words sent along channels $k+1, k+2, \dots, n$ at time t in terms of the "plaintext" words sent along channels $1, 2, \dots, k$ at time t .

The work and memory required have upper bounds (since $n \leq Q$). These upper bounds are shown in the table below:

Field	Time to precompute by parallel implementation	Number and size of processors for parallel implementation	Storage required for $SEN(k, n, Q)$
$GF(2+4) = GF(16)$	milliseconds	16 4-bit	1 k bit
$GF(2+8) = GF(256)$	seconds	256 8-bit	600 k bit
$GF(2+12) = GF(4,096)$	minutes	4,096 16-bit	300 m bit
$GF(2+16) = GF(65,536)$	hours	65,536 16-bit	70 g bit

A computer program incorporating the cool precomputation is contained in Appendix H.

2.5 The third stage of computation in the mean cases of the Bloom p/s/r process, the hot precomputation.

The receiver R is moving right along, receiving all k plaintext channels from the sender S for a while, and then some channels fail. Using means which lie outside the scope of this Phase I SBIR effort, the receiver finds at least k channels which are still operative among the n channels the sender is using. He makes a choice of exactly k of these operative channels any way he chooses, perhaps by picking the first k of them but almost certainly in a predesigned automated manner. Such a choice amounts to an injection

$$w: \{1, 2, \dots, k\} \rightarrow \{1, 2, \dots, n\}.$$

Like the sender S , the receiver R has already singled out the matrix $M[Q-n+1, Q-k]$. In practice he has trimmed off all the zero columns on its right side.

On the face of things the receiver would have to use the information contained on the injection w to set up a way of using elementary row operations to do a reduction of $MID[Q-n+1, Q-k]$ to a variant of Hermite normal form before real-time on-line decode could proceed.

This would appear to take as many as a small multiple of n^3 operations in the small k case (since the relevant matrix is $n-k$ by n). But there are artifices to reduce the computation time uniformly to yield a bound which is more like a small multiple of

$$P(n,k) = n * (n-k) * \min\{k, n-k\}$$

operations. Clearly $P(n,k) \leq (n^3)/4 \leq (Q^3)/4$, (the worst case being $k = n/2$).

Moreover $P(n,k)$ is rather small (is less than kn^2) if k is small, and is smaller still (is less than $n(n-k)^2$) if $n-k$ is small (i.e. if k is large).

The routines which achieve this improvement over straightforward linear algebraic row reductions are based on a trivial lemma, which is nevertheless worth stating.

Lemma: Let

$$\begin{aligned} \text{DATA} &= \{1, 2, \dots, n-k\} \cup \text{RANGE}(w) \\ \text{DESIDERATA} &= \{1, 2, \dots, k\} \setminus \text{RANGE}(w) \\ \text{DELEND A} &= \{n-k+1, n-k+2, \dots, n\} \setminus \text{RANGE}(w). \end{aligned}$$

Then the sets DATA and DELEND A contain the same number of members. Moreover the set DATA is disjoint from both DESIDERATA and DELEND A.

Proof: Let A be the number of members of $\text{RANGE}(w)$ which are no larger than $n-k$. In other words the set DATA contains A members. It follows that there are $k - A$ members of $\text{RANGE}(w)$ which exceed $n - k$. Hence the number of members of

$$\{n-k+1, n-k+2, \dots, n\} \setminus \text{RANGE}(w)$$

is equal to

$$[n - (n-k)] - [k - A] = A.$$

Obviously $\text{DATA} \cap \text{RANGE}(w) = \emptyset$. On the other hand DESIDERATA \cap DELEND A contains no member of $\text{RANGE}(w)$. This ends the proof.

A computer program incorporating the hot precomputation is contained in Appendix I. The idea behind Stage 3, the receiver's cool precomputation in this program is to exploit the Lemma. It enables the receiver to avoid carrying out a complete row reduction of $\text{MID}[Q-n+1, Q-k]$ to Hermite normal form. The DATA/DESIDERATA/DELEND A breakup of the set of column indices $\{1, 2, \dots, n\}$ has a partial reflection in the row indices of $\text{MID}[Q-n+1, Q-k]$. The result is that many rows are irrelevant to the production of the decode matrix COD_w described here. Moreover it is often possible to use this breakup to partition the rows of $\text{MID}[Q-n+1, Q-k]$ into three sets, one of which can be ignored, and the second of which can be used to act on the third. A careful reading of the program will also explain the bound

$$P(n,k) = n * (n-k) * \min\{k, n-k\}$$

on the number of operations, a much smaller bound than the bound n^3 which unimaginative use of standard linear algebraic techniques would suggest.

2.6 The fourth stage of computation in the mean cases of the Bloom p/s/r processes, the real-time on-line encode or decode

After finishing the third computational stage, the receiver's hot precomputation, the receiver R is ready to decode. He has a matrix REC whose rows are indexed by the set $DESIDERATA$, and which has n columns. Thus REC is no larger than a k by n matrix. Let $j \in DESIDERATA$. Then $REC(j,j) = 1$. Moreover $REC(j,k) = 0$ for every $k \in DELENDATA$. Recall that $+$ coincides with $-$ in our Galois field $GF(2^b) = GF(Q)$. It should be evident that the receiver can reclaim the word $I(j)$ which has been sent along channel j at time t from the words $H(w(g))$ (where $1 \leq g \leq k$) according to the formula

$$I(j) = \sum REC(j,w(g))H(w(g))$$

for every positive integer j belonging to the set $DESIDERATA$. The sum above is over positive integer $g \leq k$.

Similarly the sender has used his cool precalculation to produce a matrix SEN such that

$$H(j) = \sum SEN(j,g)I(g)$$

for every integer $j \in \{k+1, k+2, \dots, n\}$. The sum is over positive integer $g \leq k$.

The problem of the sender in encoding, and of the receiver in decoding, is to calculate quickly. This will be done as shown in Figure 1.6.1 above. So what remains is to multiply fast. And we can take advantage of the fact that in each of the top boxes in Figure 1.6.1 the multiplier remains fixed, though the multiplicands change with time. To carry out a multiply as fast as bits can be fed in is the goal. This can be done with systolic multipliers as shown in Appendix F.

To carry out a single GF(16) multiplication at maximum speed requires about 300 cells. To carry out 16 multiplications simultaneously requires about 4100 cells. The cells themselves consist of fewer than 10 active elements. So a very pessimistic estimate of the hardware required to carry out a GF(16) based p/s/r process is 100,000 active elements. This might require one or two programmable logic arrays.

The implementation of a GF(256) based p/s/r process would be larger. But, taking account of the fact that constant multipliers eliminate the need for flipflops in the basic cells in the implementation, we find that even GF(256) based p/s/r processes can be implemented using at most 256 PLAs. The chips for a p/s/r process involving at most 16 channels will cost less than \$100 today. For a p/s/r process involving at most 256 channels the price would almost certainly be below \$1000.

No pricing has been attempted, since no working prototypes exist. It seems likely that these cost estimates could be reduced substantially in a production mode. Other costs, such as boxes, wiring, etc. are easy to estimate.

There is an alternative approach which appears both faster and cheaper. The idea is to substitute memory for computations by storing tables of products and lists of reciprocals, perhaps even tables of quotients.

This is particularly attractive in the real-time on-line encode or decode since a single decoded channel (i.e. a single processor) keeps using the same multiplier. So each microprocessor can ask a common stored Q by Q multiplication table for a copy of the appropriate Q by 1 column corresponding to this multiply. Multiply thus becomes a memory fetch and the memory might even be resident on the microprocessor chip. A GF(16) based p/s/r process would need only $4 \times 16 = 64$ bits of memory for each microprocessor. A GF(256) based p/s/r process would require $8 \times 256 = 2048$ bits per microprocessor.

This sort of memory capacity goes for pennies. Of course, when there are $n = 65,536$ channels the picture changes. For each channel you need $16 * 65,536 = 400$ k bits of memory.

It is again worth stating explicitly that the decoding process and the encoding process are merely two variations on a theme. After cool precomputation the sender forms

$$H(j) = \sum \text{SEN}(j,g)I(g)$$

(where the sum is over positive integers $g \leq k$) to encode channel j for each $j \in \{k+1, k+2, \dots, k+n\}$.

After hot precomputation the receiver forms

$$I(j) = \sum_g \text{REC}(j, w(g))H(w(g))$$

(where the sum is over positive integer $g \leq k$) to decode channel j for each $j \in \text{DESIDERATA}$. So it suffices to describe the real-time on-line decode. The real-time on-line encode is more straightforward.

The abstract design shown in Figure 1.6.1 is the scheme which should be used. Once again we recall the need to maintain synchronization of channels in encode and decode. As in Section 1.4.1, it is easy to do.

2.7 Examples of Computations.

The programs contained in Appendices H and I have been used on examples, which are included. Appendix D gives tables of ENF for various fields $GF(Q)$ produced by the cold precomputation program in Appendix H. Appendix E contains examples of the encoding and decoding processes as carried out in Stages 2, 3 and 4 by the programs in Appendices H and I.

3. Summary of tasks, work, discoveries, recommendations and alternatives.

The contract between AFOSR and YLYK Ltd. to perform the work reported on here describes two tasks.

Task 1: Implement the heuristic procedure described in Section 6 of the proposal by means of computer programs, in order to produce explicitly the hyperfast pool/split/restitute encode and decode algorithms of the Bloom technique. Analyze the results, putting the matrices in the most convenient form, using further computer programs if needed. Determine the explicit solutions of the cases of most practical importance.

Task 2: Develop a set of design principles for the implementation in hardware of such p/s/r processes by means of an existing 16-bit microprocessor.

Mathematically, hyperfast Bloom k-out-of-n p/s/r processes break up into cases and into stages. There are four "extreme" cases. The case $k = 0$ is silly. The cases $k = 1$ (send the same message on all channels) and $k = n$ (hope that all sent messages get to the receiver) are trivial within the present state of technology. The case $k = n-1$ is trivial from a mathematical and an engineering viewpoint. But it seems important and may not be currently in use. Its implementation should be separate from the remaining "mean" cases. This implementation involves $2n - 3$ bitwise XOR gates in the format shown in Figures 1.4.1 and 1.4.2. No precomputations are required. For $Q = 4000$ channels and $k = n-1 \leq Q$ this involves fewer than 8000 gates and a phase lag (as described above) of some 12 bits.

All other cases, i.e.

$$2 \leq k \leq n-2 \leq n \leq Q,$$

are called "mean" cases in contrast to "extreme" cases. In view of the facts turned up in the narrative above we make

Recommendation 1: Concentrate first on hyperfast Bloom p/s/r processes over GF(2), GF(16) and GF(256). Over GF(2) you can implement an (n-1)-out-of-n p/s/r process for any reasonable size n. It will act on 1-bit words. Over GF(16) you can implement a k-out-of-n p/s/r process whenever

$$2 \leq k \leq n-2 \leq n \leq 16.$$

It will act on 4-bit words. Over GF(256) you can implement a k-out-of-n process whenever

$$2 \leq k \leq n-2 \leq n \leq 256.$$

It will act on 8-bit words. These three implementations will be general purpose (i.e. the boxes will allow the user to vary k and n).

Recommendation 2: Somebody who intends to use more than 256 channels should consider dedicated (i.e. k and n fixed in firmware or hardware) boxes and should try strenuously to keep the number of channels small. 4000 channels seems to be at or above the technically feasible upper bound.

For the mean cases

$$2 \leq k \leq n-2 \leq n \leq 2^{\dagger b} = Q$$

of a Bloom p/s/r process there are four stages of computation. The first two are noncritical straightforward linear algebraic reductions and we will not consider them further here, except to make

Recommendation 3: If a particular parameter setting (k,n) will be widely used (e.g. all F16s will always communicate with base by means of a 77-out-of-92 p/s/r process) then the second stage of precomputation, the sender's cool precomputation, can be dispensed with (more exactly, can be incorporated into the cold precomputation performed before the boxes are manufactured) in boxes dedicated to 77-out-of-92. The third stage of computation, the receiver's hot precomputation can be performed expeditiously.

A dedicated box can also free a participant in a battle from unnecessary attention to details. It will usually be cheaper than a general purpose box.

Recommendation 4. Maintain synchronization of parallel channels in encode and in decode. Do this by "doing something" trivial to the plaintext channels so that they acquire the phase lag associated with the channels which are encoded (or decoded) nontrivially. We have already discussed the obvious, and inexpensive, expedients which suffice to maintain such synchronization.

Recommendation 5: If it is desirable to combine p/s/r processes with cryptography or conventional error control then the following architecture should be employed. Encryption should precede p/s/r encoding which should, in turn, precede conventional error control encoding on the sender's end. By the same token conventional error control decoding should precede p/s/r process decoding which should, in turn precede decryption.

If it were built today a memory intensive ultraparallel prototype of a general purpose k -out-of- n send/receive box for $2 \leq k \leq n-2 \leq 254$ would be configured as follows. It would have about 1 mbit of ROM, broken up into 512 kbits to store a GF(256) multiplication table, 510 kbits to store ENF and 2 kbits to store a list of reciprocals of the nonzero elements of GF(256). For these purposes four 256 kbit (= 2+18 bit) ROM chips will suffice. The box would employ 256 8-bit processors, perhaps Z80s, to do the cool precomputations (when switched on send mode) as well as the hot precomputation (when switched on receive mode). There would be no logical harm, and only a small time penalty if n is over 200, in having the precomputations done as if $n = 256$, the maximum number of channels. Cool and hot precomputations would take about a second.

The real-time on-line decode would be done by $65,536 = 256 \times 256$ dedicated "dumb" processors. The processors will be arranged in 256 clusters of 256 processors. There might be as many as 256 dumb processors on one PLA chip. During a given session (i.e. for given k and n in send mode, and for given k , n and w in receive mode.)

Each processor would use a 2048 bit RAM which stored an appropriate column of the $GF(256)$ multiplication table in ROM. This RAM will have been filled by the Z80s during precomputation. The 8-bit word arriving on channel i will be split into two copies eight times so that a copy of each arriving word goes into each cluster of 256 processors on its i th channel. When a word arrives at a dumb processor the processor multiplies that word by its session constant, (i.e. treats the word as an address and outputs the contents of that address). After that the outputs from each cluster are XORed together through 8 layers as in a deeper version of Figure 1.6.1. This yields decodes or encodes for each channel. This requires 128 mbits of RAM and 65,000 (extremely) dumb units capable only of outputting the contents of an address. This configuration would require 512 RAM chips with 256 kbit capacity each. We have noted that one mbit of ROM will also be needed, as well as 256 Z80s. The dumb procesors can be parts of a PLA. Presumably some 256 PLA chips would be capable of holding the needed 65,536 processors.

The system would require shift register storage devices (perhaps 1000 cells per register) and would have to verify synchronization of inputs and impose synchronization of outputs. This would require some sort of synchronization pulses in the bit streams entering and leaving the box. A promising method is to use two voltage levels for bits and a third for synch pulses, as is standard in television transmission in the U. S.

These estimates are all on the highly pessimistic side, since detailed hardware design has not yet been undertaken.

A smaller device in which $2 \leq k \leq n-2 \leq 14$ would require sixteen 4-bit microprocessors, less than 3 kbits of ROM, 256 dumb processors and 16 kbits of RAM. Phase lag would be about 10 bits.

The splitting scheme in Figure 3.1.1 looks forbidding in two dimensions. But in three dimensions it is very simple, no matter how many channels there are. Figure 3.1.2 is a different rendering of the same process. It suggests regularity of the architecture more directly.

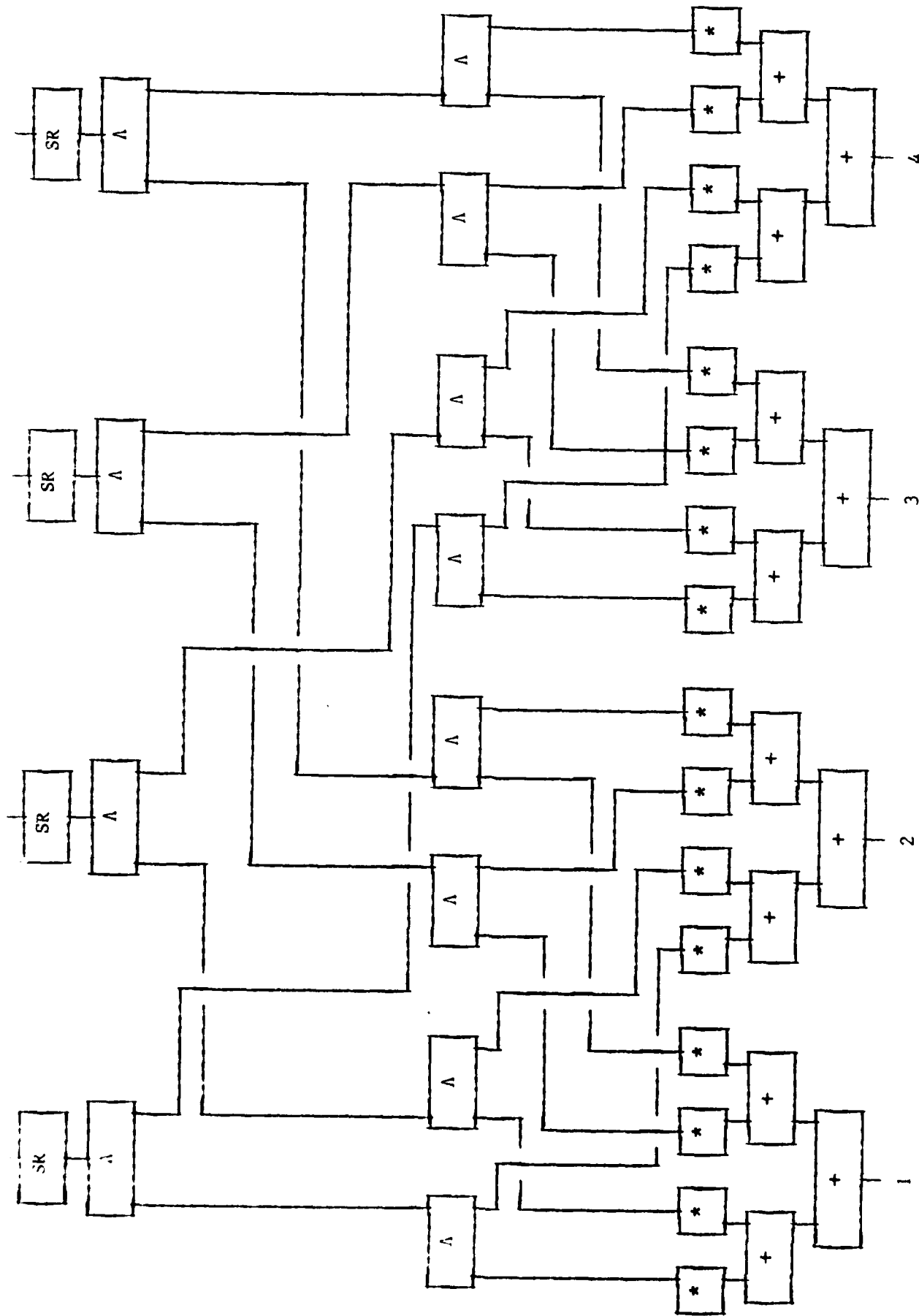


Figure 3.1.1. Abstract schematic for decode of four channels. Information flows downward.
 SR = shift register
 * = multiply successive inputs by a constant multiplier
 + = XOR of words
 Λ = take input, make two copies of it, and output one copy on each of the two outgoing lines

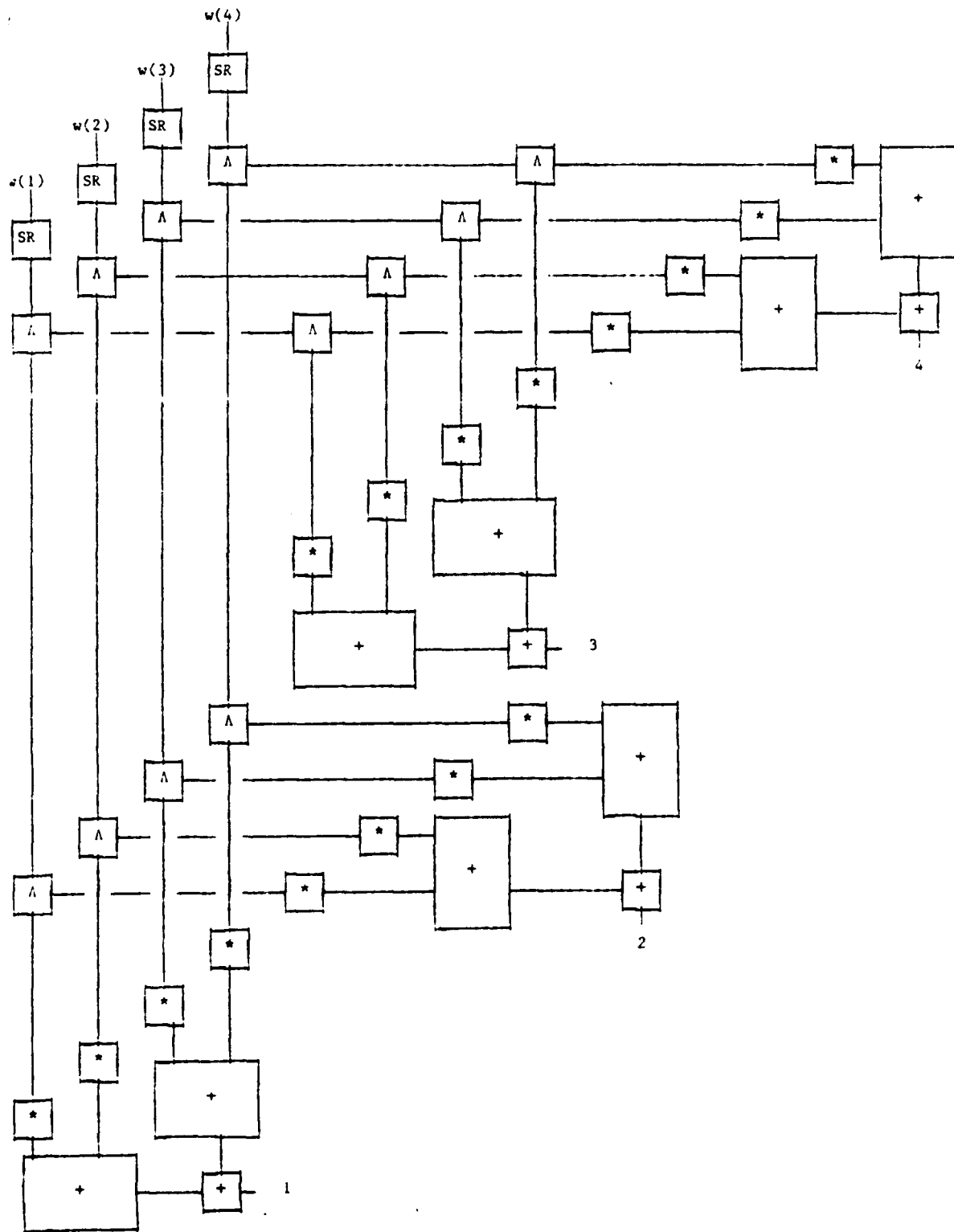


Figure 3.1.2
Alternate abstract schematic for decode of four channels

There are a number of choices facing somebody who designs hardware implementations of p/s/r processes.

YLYK Ltd. has found a very large number of ways to decode. We finally fixed on the DATA/DESIDERATA/DELENDIA approach to minimize the number of row operations at the receiver's hot precomputation stage. But other more pedestrian approaches sometimes use less computer code. In subsequent efforts, these alternative approaches should be borne in mind. Which one is used depends on what aspect of the decoding process is most important. Our approach was to minimize the time interval between discovery of what channels were inoperative, and beginning of real-time on-line decode.

There is one alternative which should be resolved as late as possible in an SBIR Phase II effort to produce a prototype. The reason for delaying a decision is the continual shift in relative costs and speeds of hardware in the marketplace. The alternative in question is whether to use computation or memory to do Galois field multiplies and divides. On the one hand there are systolic multipliers. On the other, a table of GF(16) products requires only $16 \times 16 \times 4 = 1024$ bits of memory. The table below tells the story for various fields.

Field	Number of bits to store table of products	Number of bits to store table of quotients	Number of bits to store list or reciprocals
GF(16)	$16 \times 16 \times 4 = 1024$	$16 \times 15 \times 4 = 960$	$15 \times 4 = 60$
GF(256)	$256 \times 256 \times 8 = 512k$	$256 \times 255 \times 8 = 510k$	$255 \times 8 = 2k$
GF(4,096)	202 m	202 m	50 k
GF(65,536)	69 g	69 g	1.1 m

Memory is cheap. The problem is speed. If words can be accessed quickly enough, the use of lookup for multiplication and division is attractive. XOR of words will, of course, be used for addition and subtraction.

Consider a $GF(16)$ based p/s/r process. If each of 16 4-bit microprocessors has 64 bits of memory "on-chip" the receiver's hot precompute can load the appropriate column of the multiplication table into these 64 locations on each microprocessor. This will reduce multiplication to a lookup of a 4-bit word on a list of 16 words. A $GF(256)$ based p/s/r process would need 2048 bits of memory "on-chip" available to each of the 256 processors used in real-time on-line decode. Multiply would be lookup of an 8-bit word on a list of 256 words after the appropriate column of the multiplication table had been loaded into a given processor. What we have said about real-time on-line decode applies also to real-time on-line encode, of course.

The relative merits of this approach, as opposed to a systolic system for computing products algorithmically, could change drastically as new products came onto the market or the prices of old products fell.

Another unresolved alternative concerns all three stages of precomputation. Should we use many "smart" existing processors for the precomputations or smarten up the custom designed processors used for real-time on-line encode or decode so that they can carry out the precomputations as well as the encode/decode?

Many of the cheapest old 4-bit and 8-bit processors operate below 1 mhz, whereas newer more expensive PLA can be driven faster. It would take development time to configure smart PLA to perform precomputations, whereas existing processors can be quickly programmed. It seems prudent to delay this decision as long as possible, with a view to the state of the components market the day it is made. Other choices seem more straightforward. It hardly seems worthwhile to try to fine tune field size so as to get, for example, a 17-out-of-34 p/s/r process over $GF(32)$. The simplicity of assuming that n is no larger than the field size is worth seeking. Possible exceptions to this approach can be made on an individual basis, and will likely lead to a dedicated single purpose box, such as 3-out-of-6 p/s/r process over $GF(4)$.

4. Future.

At this point, what remains is to cast the p/s/r processes into hardware. Three obvious general purpose (i.e. variable k and n) implementations would be:

1. $(n-1)$ -out-of- n , for $n \leq 1000$ using GF(2) arithmetic on 1-bit words and requiring no precomputation;
2. k -out-of- n , for $2 \leq k \leq n-2 \leq 14$ using GF(16) arithmetic on 4-bit words and requiring precomputations of a few milliseconds in (cool) Stage 2 and (hot) Stage 3;
3. k -out-of- n for $2 \leq k \leq n-2 \leq 254$ using GF(256) arithmetic on 8-bit words and requiring precomputations lasting about a second in Stage 2 and Stage 3.

It would be interesting to produce a few dedicated (i.e. fixed k and n) implementations such as:

4. 3900-out-of-4000 using GF(4,096) arithmetic on 12-bit words (in practice they would probably be the last 12 bits of 16-bit words) no Stage 2 precomputation, and a several second Stage 3 precomputation.
5. 100-out-of-4000 using GF(4,096) arithmetic, no Stage 2 precomputation and a several second Stage 3 precomputation.
6. Some half-and-half implementation, i.e. a k -out-of $2k$ for the largest value of k which would yield a tolerably short Stage 3 (hot) precomputation. Possibly a 500-out-of 1000 implementation using GF(1,024) arithmetic on 10-bit words could hold the Stage 3 precomputation down to just a few seconds.

One mathematical topic which was not targeted for the Phase I SBIR effort is dynamic reconfiguration. Suppose a sender and a receiver start out using a 200-out-of-250 p/s/r process to communicate over 250 channels which are all operative at the start. Suppose that a new

channel goes down every few seconds. It is probably possible to do the necessary reconfiguration precomputations one at a time after each failure so as to keep communications going with negligible interruptions as the receiver migrates from one set of 200 channels to another "nearby" set of 200, to another, and so on.

Careful analysis might be able to reduce the Stage 3 hot precomputation times, given that only one channel at a time goes down. The viewpoint of this proposal is that the receiver deals with $n-k$ channel failures at once.

An engineering/ergonomics consideration which will have to be tackled in Phase II, or shortly after, is the question of how the receiver will ascertain which channels have gone down. Will it be by human decision that a channel carries nothing or carries garbage? Or will it be by some automated means of sensing when a channel goes sour statistically, and is therefore presumed to be down? Or will it be by sending periodic check sequences on each channel, the idea being that their absence on the receiving end signifies channel failure? Or will still some other system be used? There are many existing protocols and algorithms to sense when a channel is or is not operational. If possible a p/s/r process box should be a module in a larger system. This architecture would enable the user in the field to decide which method of sensing inoperative channels is appropriate to the system in use.

Such considerations may or may not influence the p/s/r hardware directly, but will certainly be important in the context in which a p/s/r process is imbedded. Matters of this sort will be taken up in more detail in YLYK Ltd's SBIR Phase II Proposal to AFOSR. Up to now speed has been the dominant consideration. In Phase II cost will come more to the fore.

Appendix A

The technical part of the YLYK Ltd. proposal
which led to this contract

U.S. DEPARTMENT OF DEFENSE
SMALL BUSINESS INNOVATION RESEARCH PROGRAM
PHASE I—FY 1983
PROJECT SUMMARY

FOR DOD USE ONLY

Program Office	Proposal No.	Topic No.
----------------	--------------	-----------

TO BE COMPLETED BY PROPOSER

Name and Address of Proposer

YLYK Ltd.
PO Box 7966
Ann Arbor, Michigan 48107

Name and Title of Principal Investigator

Mr. Bob Blakley
President, YLYK Ltd.

Title of Project

High-speed low-cost ways to get messages from a sender to a receiver when some channels linking them become inoperative.

Technical Abstract (Limit to two hundred words)

Military communications systems are subject to trauma. Certain channels fail for protracted periods of time. The red noise problem arises when some, but not all, of the channels linking a sender to a receiver become inoperative. The solutions to this problem are called pool/split/restitute processes. P/s/r processes amount to ways to encode digital messages at a sending node so as to make sure that all transmitted information gets through and is decoded correctly at the receiving node whenever at least k out of the n channels linking those two nodes remain operative. P/s/r processes are designed to work even though the sending node has no way to tell which of the channels it is using are inoperative.

It has been known for at least two years that the encode and the decode operations in a p/s/r process are faster and simpler than those in any but the weakest and most trivial error correcting codes. Moreover the bandwidth expansion is typically smaller in a p/s/r process than in an error correcting code adapted to do the same job. This project is aimed at producing a further orders-of-magnitude improvement in the theory of p/s/r processes. This carries over into a comparable improvement in implementing them.

Anticipated Benefits/Potential Commercial Applications of the Research or Development

The availability of best-possible p/s/r processes to solve the red noise problem will make it cheap and easy to design fault-tolerant or fail-safe communications systems at all levels of complexity, from the microscopic to the global. The ability to overcome the unpredictable permanent failure of a certain specified proportion of the channels of communication in a system may have major consequences in chip layout, design of wiring within military platforms, commercial vehicles, telecommunications networks, and global C³I structures. The speed and simplicity of the implementation of p/s/r processes gives promise of widespread cheap channel-failure insurance in gigabit per second communications.

3. Identification and significance of the problem/opportunity.

This proposal deals with research and development work on the red noise problem [AS82].* It is one facet of the message gap [BR81; AN83]. It is associated with the difficulty experienced by two or more centers in communication with one another when a catastrophic long-lasting failure of some of the communication channels linking them occurs.

More specifically, the red noise problem concerns a sending node and a receiving node linked by several parallel channels over which information is moving in digital form. The problem is this. Suppose you are prepared to accept the failure of $n-k$ out of the n channels which are initially functioning. How do you encode the information at the sending node so that all of it gets through as long as any k channels remain operative? How do you decode this information at the receiving node? Ways of doing this are called pool/split/restitute processes.

Examples of systems faced with the red noise problem are numerous. A few of them are:

- I. Within a single vehicle or platform -- such as a missile, an aircraft, a ship, a tank or a spacecraft -- there might be eight separate wires or fibers carrying information from an area containing power supplies, engines, control devices and weapons to an area containing human or electronic controllers. It is imperative that the controllers continue to receive all of the highest priority types of information even though three wires (nobody knows in advance which three) or fibers are cut by accident or trauma. This guaranteed 5 out of 8 reliability may have to be cheap in the sense that it must be provided by tiny inexpensive circuitry;
1. At the global level or the theater level, consider communications between commanders and subordinates, or between separate command centers (whether these are vehicles or cities or redoubts or satellites is irrelevant mathematically) connected by ten communications channels. Several of these channels might be optical fibers, several might be microwave relay tower chains, and a few might be satellite relay links. In the event of emergency it might be imperative for all high level communications to get through continuously after six of these ten channels fail, even when the sender does not know which four of his outgoing channels are successfully carrying their information to the intended receiver. It might be imperative to provide this guaranteed 4 out of 10 reliability to communications systems working at very high bit rates;
- III. On the microscopic scale, VLSI and VHSIC are forcing more active elements and more pathways onto a chip. It is increasingly important to assure the safe arrival of every bit at the proper place in timely fashion even though certain circuit elements fail. This must be done in an extremely simple way so as not to gobble up too much of the chip just for this assurance of reliability. Perhaps it would be desirable to use an 8 out of 10 p/s/r process to move a 16-bit word from memory along ten 2-bit channels so that the whole word gets through despite the failure of any two of those ten channels.
- IV. The word "channel" should not be allowed to obscure the abstract possibilities. Separate packets in a local area network can be treated as separate channels since each packet can be 500, 1000, 2000 or some such large number [ST83] of bits. The bits in a single packet get through all together

*Footnote: All entries in square brackets refer to the bibliographic citations list beginning on page 17.

or not at all, according as the packet reaches its destination, or else is destroyed in a collision or otherwise goes astray [BL83a, p. 5; P082, pp. 76-101]. If collisions and misroutings are present, but rare, a 63 out of 64 p/s/r process applied to successive batches of 63 packets from a given sender to a single receiver might provide cheap insurance at a bandwidth expansion of $1/64 = 1.5\%$.

Obviously, comparable examples could be produced in many other contexts. But abstractly they all point up the same need. It is important to find extremely simple encode/decode schemes to provide cheap ways of assuring very high bit rate solutions to the problem of getting all the important information from sender to receiver whatever channels remain -- in the absence of prior (or even concurrent) knowledge of which channels are the lucky survivors -- as long as there are enough channels still operative to come up to the initial specifications.

This might sound reminiscent of the use of error correcting codes to correct burst errors, and in a way it is. However, during the two years since the red noise problem was recognized [AS82] as important in its own right, tailor-made solutions have been advanced which are much cheaper (algorithmically, but this entails a comparable dollar saving in implementation) than, and much faster than, the use of standard error correcting code techniques [BL83a, pp. 367-389; MC77, pp. 181-186, 212-213; BE68, pp. 393-394; VI79, pp. 227-300] to solve it.

A moment's reflection shows why this might be so. Error correcting codes are designed to deal with errors occurring anywhere in the transmitted data stream (as long as these errors are not too numerous) [VI79, p. 34]. These errors can be very irregularly spaced. In a mathematical sense which should become clearer below, red noise errors can be viewed as occurring with a definite periodicity in the received bit stream. Such a well behaved type of error, of course, constitutes a subproblem of the general error correction problem. So it seems plausible (and turns out actually to be the case) that the solution might be conceptually simple, as well as easy to implement in a cheap fast way. The recent literature [AS82] and some as yet unpublished work, bears this out. But in 1983 a further remarkable simplification and speedup of both the encoding and decoding processes used to solve the red noise problem has been suggested by current research. Several important instances of this further orders-of-magnitude improvement have been discovered and verified as the result of a powerful heuristic principle. The research on this project will attempt to turn this heuristic principle into a rigorous tool for producing this orders-of-magnitude improvement of both the speed and the cost of the encoding/decoding schemes for combatting red noise in many or all cases of the problem. It aims to produce a complete taxonomy of best possible (or, more properly speaking, almost best possible) solutions of the red noise problem. Time permitting, it will make a preliminary abstract analysis of how to design electronic implementation of these coding/decoding processes using cheap off-the-shelf components to attain bit rates well above a megabit per second.

4. Background, technical approach and anticipated benefits.

4a. Background. An understanding of the red noise problem and the objects which solve it, namely pool/split/restitute processes, is best acquired by looking at the history of the last five years. In a 1978 NSF proposal, Blakley invented a new cryptographic object, the threshold scheme (He called it a key safeguarding scheme, but Denning's well known cryptography and data security textbook [DE82] has made threshold scheme the standard terminology). His paper describing the notion, and giving the first example was presented at NCC '79 and published [BL79] in the proceedings of that meeting.

A k out of n threshold scheme is a mathematical way of utilizing a source of random bits to take an important piece of digital information, called a substance

(there isn't much harm in thinking of a substance as just being a plaintext message) and produce n output pieces of information called shadows of the original substance. A shadow can, without too much inaccuracy, be thought of as being part of a ciphertext message. Every shadow is about the same size as the substance and, collectively, the shadows securely carry the full import of the substance in the following sense. There is, on the one hand, a trivial algorithm which can reproduce the substance if any k of the n shadows are inputted to it. But, on the other hand, it is impossible to gain any inkling of the value of the substance on the basis of knowledge of only $k-1$ or fewer of the shadows. The justification of this latter statement is somewhat technical. Nevertheless the basic idea can be expressed fairly briefly in terms of what Konheim [K081, p. 31] calls the Bayesian opponent. Just as it is possible to prove [BL81a] the one-time pad [DI79 pp. 399-400, DE82 pp. 86-87] perfectly secure in the Shannon [SH49] sense, so it is possible to prove that a k out of n threshold scheme is (Shannon) perfectly secure up to threshold k . This means that the Bayesian opponent cannot modify a (perhaps shrewd) initial guess regarding the substance on the basis of knowledge of only $k-1$ shadows. Somewhat more formally:

A posteriori probability that the substance has a value equal to S (given that the objects $h(1), h(2), \dots, h(k-1)$ are known to be shadows of that substance)
 = A priori probability that the substance has a value equal to S .

To be more concrete, suppose there is a roll of magnetic tape (the substance) which contains the full inventory of payloads, locations and targets of all missiles belonging to A on day b . Somebody might think this information important enough to merit protection by a 4 out of 9 threshold scheme. This will involve use of a trivial algorithm which takes this original roll of tape, together with 4 tape rolls worth of random bits, and produces 9 rolls of mag tape (the 9 shadows of the original substance) as outputs. Now an opponent of A , let us call it R , might quite correctly suspect at the outset that several of these missiles are targeted on some important spot, call it M . But if R can only obtain 3 of the (shadow) rolls of mag tape it cannot shed any new light on this initial conjecture. It started out with a good bet that its conjecture is correct. It winds up with exactly the same odds. If R can get 4 of the 9 rolls, of course, the game is over. It has crossed the threshold of information and can reconstruct the entire original roll of mag tape. So it knows everything A does.

Shamir, by the way, introduced the threshold terminology in a paper [SH79] which independently invented the idea of threshold scheme a few months after [BL79], and gave a better example of how to implement the notion. After the Blakley [BL79] and Shamir [SH79] papers appeared, several people interested in information theory and computer science took up the topic. Asmuth and Bloom [AS81] produced a huge family of threshold schemes, of which Shamir's was a special case. They also gave the only way known to date for "spoofproofing" a threshold scheme, a notion we won't consider further here. But they paid a price for this extra feature, a small departure from Shannon perfect security. Then Bloom [BL81b] generalized the one-time pad (really the 2 out of 2 case of a threshold scheme, rather than a true [BL80; DE82, p. 152] cryptosystem) so as to produce essentially the fastest possible threshold scheme. He also noted that it is possible to reduce message expansion in a threshold scheme, but only at the cost of reducing security.

Blakley [BL79], Shamir [SH79], Asmuth and Bloom [AS81], and Bloom [BL81b] independently discovered that any k out of n threshold scheme which made use of a finite field [JA64, pp. 58-62; PL82, pp. 44-58; BL83a, pp. 65-92] required that the field contain at least n elements. Bloom gave a persuasive argument [BL81b] to the effect that this was necessary in order to attain Shannon perfect security. David, DeMillo and Lipton [DA80] produced another threshold scheme. Hellman, in company with his students Karnin and Green [KA81], produced schemes without sharp thresholds and

showed that adding certain desirable features to threshold schemes necessarily impairs Shannon perfect security, thus explaining what Asmuth and Bloom [AS81] had observed regarding spoofproofing. McEliece and Sarwate [MC81] produced yet another threshold scheme, and drew the theories of threshold schemes and of error correcting codes into a single compass by exhibiting an explicit relationship between Shamir's [SH79] scheme and Reed-Solomon codes [RE60; BE74, pp. 70-71].

Two aspects of threshold schemes worth noting explicitly are:

- I. Threshold schemes are related to error correcting codes. But the "decode" in a threshold scheme is trivial, whereas decode can be a formidable [BE78; NT81] problem, even an NP-complete [GA78] problem, in an error correcting code.
- II. As of 1982, most k out of n threshold schemes made use of finite fields (Galois fields) [JA64, pp. 58-62; MA77, pp. 93-124; PE72, pp. 144-169]. All [AS83; BL79; BL81b; SH79] such schemes required a field with at least n elements.

Last year, Asmuth and Blakley [AS82] explicitly enunciated the red noise problem and solved it by means of a p/s/r process based on the Chinese Remainder Theorem. This p/s/r process could be viewed as being just "an Asmuth-Bloom threshold scheme completely lacking in cryptographic security". Its great advantage was its flexibility in dealing with information sources with very different bit rates. Years ago Stone [ST63] had used much the same approach to solve a problem in the theory of error correcting codes.

4b. Technical approach. With this background it is now possible to give the general framework of the present research. The principal investigator, Bob Blakley, has already taken a Bloom threshold scheme and produced from it the corresponding p/s/r process. It will be called, simply, a Bloom p/s/r process below. He has simulated its operation on a high speed digital computer.

The k out of n case of this Bloom p/s/r process works as follows. Suppose that b is a whole number (positive integer [MA67, p. 47]) so big that $2^b \geq n$. Then any ancestral list $(a(1), a(2), \dots, a(k))$ of k words [MA67, p. 43] (each of which is a b -bit word) is turned into a descendant list $(d(1), d(2), \dots, d(n))$ of n b -bit words. This is the encode (i.e. the pool/split) process. It is done in such a way that any k -word sublist [MA67, p. 228] $(d(j(1)), d(j(2)), \dots, d(j(k)))$ of the descendant list $(d(1), d(2), \dots, d(n))$ contains enough information to reclaim the ancestral list $(a(1), a(2), \dots, a(k))$ in its entirety. This is done by a decode (i.e. reconstitute) process which uses no more than trivial linear algebra over the finite field $GF(2^b)$. By comparison with threshold schemes and error correcting codes this Bloom-style p/s/r process has the following features.

- I. Its k out of n case effects only (n/k) -fold message expansion. Thus its 8 out of 10 case effects a 25% message expansion (from 1 unit to $10/8 = 1.25$ units). This expansion is quite obviously best possible for a scheme which can recover eight b -bit ancestral words from any eight of ten b -bit descendant words.
- II. The Bloom p/s/r is, to all intents and purposes, the p/s/r process which uses the smallest possible number of arithmetic operations in the finite field it utilizes. Its "encode" (i.e. pool/split) and "decode" (i.e. reconstitute) processes are both trivial, exhibiting much less computational complexity than the decodes in any error correcting code which might be adapted to do the same job. The reason for this is that the error correcting code exhibits overkill because it is a general purpose tool. It is invented to deal with many more types [HA80, p. 24] of "errors" than one encounters when dealing

with red noise. This p/s/r process is a special-purpose tool for dealing with red noise.

- III. P/s/r processes are not cryptographic objects in any sense of the word. They do not involve any type of cryptosecurity. They do nothing more than guard against loss of signal, and therefore fall within the general area of error control.

4c. Anticipated benefits. The linear algebra of large finite fields can take many machine cycles per multiply or divide. It can also, in the worst circumstances, make considerable demands on memory. During 1983 a heuristic principle has come to light which massively reduces this aspect of the computation in numerous cases. From Bloom p/s/r processes it produces hyperfast p/s/r processes which encode and decode bytes or larger words in less than ten machine cycles (on highly parallel processors) for almost all practical choices of k and n . This heuristic suggests the possibility of comparable reductions in many other cases. Consider an example which at first blush seems extreme. In April, 1983 we have reduced the memory requirements for one implementation of a 60 out of 62 scheme by orders of magnitude. As regards the parameters, 60 out of 62, one cannot readily conceive of so many fibers joining two nodes. But, returning to the packet-switching example above, it is easy to imagine one or two packets out of sixty going astray. Also, recently developed continuously reconfiguring multimicroprocessor control systems [EL83] appear to have many virtual channels.

At any rate it appears that this heuristic principle -- already successful in making a k out of $k+1$ or a k out of $k+2$ Bloom p/s/r process capable of decoding in something like $3k$ machine cycles on an ordinary microprocessor, and in about $\log(k)+2$ cycles on a parallel processor -- will lead to ways to reduce the run time of hardware implementation of all k out of n schemes by comparable amounts. This should make them able to run on gate arrays, programmable logic arrays or other standard cell [NE83, pp. 470-471] hardware, or even other cheap off-the-shelf devices, at rates well above the megabit per second range.

The ability to code and decode at such bit rates becomes increasingly desirable with the emergence of tiny cheap cleaved coupled-cavity lasers [TH83]. They make it possible to use a 73 mile fiber without a repeater [AB83] to communicate at 420 megabits per second with an error rate of 10^{-9} [TH83; LI83, p. 363]. It seems likely [GO83] that terabit per second communication systems are in the offing now that 30 femtosecond light pulses are available. Theoretically, further orders-of-magnitude improvements in processing gains because of exploitation of photonic efficiency of detectors [GA83, p. 526] as well as by means of preservation of polarization [RA83] are possible even after that. Until optical computers are developed we will need code/decode schemes of minuscule computational complexity to deal with such bit rates.

The hyperfast p/s/r processes have a further advantage, in addition to low computational complexity (which amounts to high-speed low-cost implementability on simple hardware). They can also be implemented in a highly parallel way, so that separate devices can do concurrent decoding for separate channels, and each device can do many operations in parallel.

It is now clear how to move digital information with minimum redundancy and maximum speed (an unusual plus, best possible in two ways) at a modest dollar cost (which does, however, rise with desired data throughput rate) so as to overcome a predetermined level of threat of channel failure.

Presumably the existence of such a capability could affect the design of everything from chips to the fiber "wiring" of missiles, ships, tanks, planes and the

design of C^3I systems in the future. It certainly has to implement the military digital switching systems criteria [R083, p. 19] of survivability, endurability, distributed communications, responsiveness, efficient spectrum utilization, and cost effectiveness. Thus this proposal arguably addresses 6 of the 9 military operational system requirements detailed in [R083, p. 19].

4d. Foundations for Phase II. By the end of Phase I all the algorithmic principles needed to encode and decode in a hyperfast p/s/r should be known for every n and k . Basic principles of design for hardware implementation of these algorithms should also be available. The design principle will lead one way if minimum cost is the ultimate goal, another way (high parallelism and custom design) if ultrahigh speed is the overriding aim, and still a third way if a single unit is to be used for various different values of k and n .

But in any case the abstract basis on which to proceed to build bench systems, and then prototypes of field systems, will be firmly in place. The actual design and testing program can begin as soon as Phase I is complete.

5. Phase I technical objectives.

In Phase I we hope to exploit the heuristic described in Section 6 below to use the Bloom approach to suggest a new collection of p/s/r processes, the hyperfast p/s/r processes. The k out of n case can be expected to reconstitute (i.e. decode) 15 bits of the information contained in one input channel using fewer than $3k$ machine cycles on an existing 16-bit microprocessor if $n - k$ is smaller than 16. The memory requirement for table lookup implementation will be well under 1000 bytes.

The various channels can be recovered concurrently on separate machines if so desired. Since the decode process is simply a linear combination $\sum c(i)d(i)$ of received words $d(1), \dots, d(k)$ with fixed coefficients $c(1), \dots, c(k)$ it is even possible to design a vector microprocessor machine which can move $2k$ words concurrently, then perform k products by table lookup concurrently, then add (which is just XOR in $GF(2^b)$, and thus has no carry propagation) k summands in a single operation. The vector fetch and the vector dot product $\sum c(i)d(i)$ can, in theory, be done in 1 cycle each. The XOR of k summands can be done in $\log(k)$ cycles or fewer (the log being to base 2). It is even possible to use VLSI to produce a cheap ultraparallel implementation in terms of hardwired functions with more than two inputs if n is not too large.

Examples of times to reconstitute 15 bits on one channel in implementing the k out of n case of such a hyperfast p/s/r process appear to be:

k	n	number of cycles (ordinary microprocessor implementation)	number of cycles (k -vector microprocessor implementation)	number of cycles (ultraparallel implementation)
1	10	3	3	4
2	10	6	3	4
4	10	12	4	4
8	10	24	5	4
16	30	48	6	4
32	40	96	7	4
64	70	192	8	4
128	140	384	9	4
256	270	768	10	4
512	520	1536	11	4
1000	1019	3072	12	4

The expected form of the encode algorithm is so similar to the expected form of the decode that we will not discuss it here. See Section 6 below.

The first technical objective of Phase I, then, is production of the encode algorithm and the decode algorithm for the k out of n case of a hyperfast p/s/r process. Each of these is in the form of a bunch of separate and independent dot products in k or n dimensional vector spaces over $GF(2^b)$ for some positive integer b near $\log(k)$.

The second technical objective is a portfolio of abstract design principles for implementation of such a p/s/r process. YLYK Ltd. plans to sketch the abstract principles behind implementing such a k out of n p/s/r process by means of an existing 16-bit microprocessor, an existing programmable logic array or gate array, and a hypothetical vector microprocessor with a 16-bit word size, and vectors of up to 1024 words.

It is to be emphasized that the plan for Phase I is to deliver the encode and decode algorithms in definitive and final form. But YLYK Ltd. will only sketch, as time allows, the basic abstract features of hardware implementation. YLYK Ltd. will not produce hardware, or even the final design of hardware, in Phase I.

6. Phase I work plan.

It is no longer possible to avoid technicalities. Before we describe the heuristic device for producing these cases of hyperfast p/s/r processes and, thereafter, finding the general hyperfast p/s/r process it is necessary to look more deeply into the geometry of Bloom p/s/r processes. The collection

$$V(k, F) = \{(1, m, m^2, m^3, \dots, m^{k-1}) \in F^k : m \in F\}$$

is in general position [YA68, p. 164; MA77, p. 326] in the k -dimensional vector space F^k over any field F . In other words, suppose that k lies between 2 and the cardinality [MA67, p. 53] of F . Then every k by k matrix of the form

$$\begin{bmatrix} 1 & m(1) & m(1)^2 & \dots & m(1)^{k-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & m(k) & m(k)^2 & \dots & m(k)^{k-1} \end{bmatrix}$$

(where the $m(i)$ are pairwise distinct) is nonsingular because it is a Vandermonde matrix [H071, p. 125]. The formal definition, then, is that a set of vectors is in general position in a k -dimensional vector space W if every one of its k -member subsets is a basis for the space W . More important than what we said about $V(k, F)$, but far less trivial, is the fact that

$$V^*(k, F) = V(k, F) \cup \{(0, 0, \dots, 0, 1, 0, \dots, 0)\} = V(k, F) \cup \{e\}$$

(where the 1 is in any position) is also in general position. This requires use of the theory of symmetric polynomials [RE67, pp. 457-458]. So getting just one more vector into the set takes a lot more doing. But so far the extra effort seems essential to what we propose to do. The way a Bloom k out of n p/s/r process works is to take a fairly large set of vectors (at least n of them) in general position in the k dimensional vector space $GF(q)^k$ over the field $GF(q)$ of q

elements. There's no harm in taking $V^*(k, GF(q))$ if $q \geq n$. Suppose that q is a power of 2, i.e. that $q = 2^b$. Suppose, also, that the p/s/r process is meant to work by accepting one b-bit word after another from each of k input channels (ancestral channels) at the source. It should then send one b-bit word after another down each of n descendant channels to the receiver. Each one of these descendant channels is identified with a vector belonging to $V^*(k, GF(2^b))$. Once some channels fail, and a decoding scheme is employed on k of the channels which still work, it acts the same way on every successive b bits in each channel. So it suffices to look at a single time slice through the system. In such a slice encoding is done by defining a linear map [H071, p. 67] $L : GF(2^b)^k \rightarrow GF(2^b)$ by setting

$$L(w(i)) = \text{the } i\text{th } b\text{-bit ancestral message}$$

for the vectors $w(1), w(2), \dots, w(k)$, in some ordering of $V^*(k, GF(2^b))$, which corresponds to the k ancestral inputs. These are assumed to be sent unaltered down the first k descendant channels. In addition to that, the sender solves for any other member y of $V^*(k, GF(2^b))$ in the form

$$y = c(y,1)w(1) + \dots + c(y,k)w(k)$$

as a linear combination of the $w(i)$ with coefficients $c(y,i)$ drawn from $GF(2^b)$. Down the channel corresponding to y is sent the message

$$Ly = L(c(y,1)w(1) + \dots + c(y,k)w(k)) = c(y,1)Lw(1) + \dots + c(y,k)Lw(k).$$

Addition is $GF(2^b)$ addition (i.e. exclusive or, XOR, of b -bit words) and multiplication is $GF(2^b)$ multiplication, since both $c(y,i)$ and $Lw(i)$ are members of $GF(2^b)$. All the linear algebra is a precomputation, of course. Hence the $c(y,i)$ are available before encoding starts. Decoding involves a once-for-all solution (another precomputation) of linear equations to find the $\{w(1), w(2), \dots, w(k)\}$ in terms of a collection of any k of the y 's. This gives the $Lw(i)$'s (the ancestral b -bit messages) in terms of the Ly 's (the descendant b -bit messages). The whole thing works because any k members of $V^*(k, GF(2^b))$ are a basis for the vector space $GF(2^b)^k$, i.e. because of the general position assumption.

This sounds abstract, for the usual reason. It was written to fit into a small compass, without too many numbers and subscripts littering the printed page. But all the objects are explicitly given. For example, a 3 out of 7 p/s/r process could make use of the field $GF(8)$, the 3-dimensional vector space $GF(8)^3$, and the 9-member set

$$V^*(3, GF(8)) = \{(1, m, m^2) : m \in GF(8)\} \cup \{\epsilon\},$$

where ϵ is either $(0,1,0)$ or $(0,0,1)$. For a Bloom p/s/r process it doesn't matter which. For our purposes, building hyperfast p/s/r processes, the choice of ϵ seems to be crucially important. It appears to require an amount of trial and error tedious for humans, but trivial on a computer.

A k out of n Bloom threshold scheme would require use of $GF(2^t)$ where $2^t \geq n$. Thus a 990 out of 1000 scheme would require $GF(1024)$ multiplications. In table lookup mode this would require a table of over one million 10-bit words.

Obviously one would trade time off against memory. But then each multiplication would involve dozens of machine cycles, and each division could require hundreds. The simple heuristic we describe below says that the threshold scheme analogy is hopelessly pessimistic. A 990 out of 1000 hyperfast p/s/r process should require only GF(16) multiplications. This uses only a table of 256 four-bit words.

The heuristic for producing hyperfast k out of 2^{b+1} p/s/r processes which use linear algebra over extremely small fields of characteristic two [JA64, p. 61; PL82, p. 46; BL83a, p. 80] goes as follows. Do not use just any collection of 2^{b+1} vectors in general position over $GF(2^b)^k$. Use $V^*(k, GF(2^b))$, where the vector $\epsilon = (0, 0, \dots, 0, 1, 0, \dots, 0)$ is chosen by trial and error from among the k possible unit coordinate vectors [NO69, pp. 473-474] in $GF(2^b)^k$ to satisfy the following condition.

Heuristic: A k out of $k+j$ hyperfast p/s/r process can be formed, in the Bloom manner, over $GF(2^b)$ if $j < 2^b$. Form a Bloom p/s/r process using $V^*(k, GF(2^b))$ for each possible choice of $\epsilon = (0, 0, \dots, 0, 1, 0, \dots, 0)$ and examine the corresponding coefficients $c(y, i)$. There is a minimal ϵ , in the sense that all the $c(y, i)$ for this ϵ belong to a smallest subfield of $GF(2^u)$, where $2^u \geq k+j$. This minimal ϵ may have the property that all $c(y, i)$ belong to $GF(2^e)$, where $j < 2^e$, and where e is the smallest integer exponent for which this is true.

In the following paragraphs we will give some motivation for the heuristic. Here is a summary of the known cases of a hyperfast p/s/r process it has suggested, directly or indirectly:

- 4 out of 5 over GF(2), followed by general k out of $k+1$ over GF(2);
- 3 out of 6 over GF(4), and 4 out of 6 over GF(4);
- 7 out of 14 over GF(8).

This last was made possible, with limited computer power, by adroit use of Zech's logs [MA, p. 91-92]. It might lead to a more general k out of $k+7$ hyperfast p/s/r process over GF(8) soon. Conceivably the cases 8 out of 14, 9 out of 14, and 10 out of 14 can also be produced over GF(8) and made to give rise to more general cases involving k out of $k+7$, k out of $k+6$, k out of $k+5$ and k out of $k+4$ over GF(8). But to get such things as a k out of $k+8$ p/s/r process using only the arithmetic of GF(16) will likely require the effort and the computer power of an IBM PC programmed in assembly language running for hours.

We recall that $GF(2) = \mathbb{Z}/(2)$ is [BL83a, pp. 69, 75] the field of two elements, i.e. the integers modulo 2, i.e. the set $\{0, 1\}$ under the addition and multiplication tables

+	0	1
0	0	1
1	1	0

*	0	1
0	0	0
1	0	1

The following encode and decode rules obviously work for a 4 out of 5 p/s/r, where all arithmetic is done in GF(2). To encode (i.e. to pool/split) an ancestral list $(a(1), a(2), a(3), a(4))$ of four 1-bit words, let

$$\begin{aligned} d(1) &= a(1); & d(2) &= a(2); & d(3) &= a(3); & d(4) &= a(4); \\ d(5) &= a(1) + a(2) + a(3) + a(4). \end{aligned}$$

To decode if $d(5)$ is missing set:

$$a(1) = d(1); \quad a(2) = d(2); \quad a(3) = d(3); \quad a(4) = d(4).$$

If $d(1)$ is missing set;

$$a(1) = d(2) + d(3) + d(4) + d(5); \quad a(2) = d(2); \quad a(3) = d(3); \quad a(4) = d(4).$$

If $d(2)$ or $d(3)$ or $d(4)$ is missing the obvious analog of the case immediately above decodes successfully. This can be more readily seen in terms of matrices over [H071, p. 6] the field $GF(2)$

$$A = \begin{bmatrix} a(1) \\ a(2) \\ a(3) \\ a(4) \end{bmatrix}, \quad D = \begin{bmatrix} d(1) \\ d(2) \\ d(3) \\ d(4) \\ d(5) \end{bmatrix}, \quad E = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \quad M[1] = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \quad M[2] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix},$$

$$M[3] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \quad M[4] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix}, \quad M[5] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Then encoding is the rule $D = EA$. And decoding is the rule $A = M[i]D$ when $d(i)$ is missing. This works because

$$M[i]D = M[i](EA) = (M[i]E)A = IA = A$$

when the i th entry of D is absent. This is because every $M[i]$ is a left inverse [N069, p. 11] of the nonsquare matrix E , and because $M[i]D$ is independent of $d(i)$ (since the i th column of $M[i]$ contains only zeros). Clearly [N069, pp. 11-17, 132-135] E cannot have a right inverse [N069, p. 11].

Instead of a 4 out of 5 p/s/r we could as easily have defined a k out of $k+1$ p/s/r process using only the arithmetic of $GF(2)$. This is quite unlike what happens when threshold schemes are involved. To implement a k out of $k+1$ threshold scheme you must use the arithmetic of the much larger field $GF(Q)$, where $Q \geq k+1$.

The extreme simplicity of this k out of $k+1$ p/s/r process (its use of only $GF(2)$ arithmetic) is not a fluke. Moving up the scale, it is possible to implement a k out of $k+3$ hyperfast p/s/r process using only the arithmetic of $GF(4)$. This is, one recalls [MA77, p. 101; BL83a, p. 75], the set $\{0, 1, r, s\}$ under the addition and multiplication tables

+	0	1	r	s
0	0	1	r	s
1	1	0	s	r
r	r	s	0	1
s	s	r	1	0

*	0	1	r	s
0	0	0	0	0
1	0	1	r	s
r	0	r	s	1
s	0	s	1	r

It is commonplace to represent these four "numbers" as 2-bit words:

$$0 = (0, 0); \quad 1 = (0, 1); \quad r = (1, 0); \quad s = (1, 1).$$

Evidently, then, $+$ is just the 2-bit word exclusive or operation, XOR. And $*$ can be implemented by means of a table with sixteen 2-bit entries.

For brevity we merely give the matrix form of a 3 out of 6 hyperfast p/s/r process in terms of matrices over [H071, p. 6] the field $GF(4)$.

$$A = \begin{bmatrix} a(1) \\ a(2) \\ a(3) \end{bmatrix}, \quad D = \begin{bmatrix} d(1) \\ d(2) \\ d(3) \\ d(4) \\ d(5) \\ d(6) \end{bmatrix}, \quad E = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & r & s \\ 1 & s & r \end{bmatrix}, \quad M[1,2,3] = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & s & r \\ 0 & 0 & 0 & 1 & r & s \end{bmatrix}, \quad M[1,2,4] = \begin{bmatrix} 0 & 0 & 1 & 0 & s & r \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix},$$

$$M[1,2,5] = \begin{bmatrix} 0 & 0 & s & r & 0 & s \\ 0 & 0 & r & s & 0 & s \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \quad M[1,2,6] = \begin{bmatrix} 0 & 0 & r & s & r & 0 \\ 0 & 0 & s & r & r & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \quad M[1,3,4] = \begin{bmatrix} 0 & 1 & 0 & 0 & r & s \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix},$$

$$M[1,3,5] = \begin{bmatrix} 0 & r & 0 & s & 0 & r \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & s & 0 & r & 0 & r \end{bmatrix}, \quad M[1,3,6] = \begin{bmatrix} 0 & s & 0 & r & s & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & r & 0 & s & s & 0 \end{bmatrix}, \quad M[1,4,5] = \begin{bmatrix} 0 & s & r & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix},$$

$$M[1,4,6] = \begin{bmatrix} 0 & r & s & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \quad M[1,5,6] = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \quad M[2,3,4] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & r & s \\ 1 & 0 & 0 & 0 & s & r \end{bmatrix},$$

$$M[2,3,5] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ s & 0 & 0 & r & 0 & 1 \\ r & 0 & 0 & s & 0 & 1 \end{bmatrix}, \quad M[2,3,6] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ r & 0 & 0 & s & 1 & 0 \\ s & 0 & 0 & r & 1 & 0 \end{bmatrix}, \quad M[2,4,5] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ r & 0 & s & 0 & 0 & r \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix},$$

$$M[2,4,6] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ s & 0 & r & 0 & s & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \quad M[2,5,6] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \quad M[3,4,5] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ s & r & 0 & 0 & 0 & s \end{bmatrix},$$

$$M[3,4,6] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ r & s & 0 & 0 & r & 0 \end{bmatrix}, \quad M[3,5,6] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}, \quad M[4,5,6] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

To encode, set $D = EA$. To decode, set $A = M[w,x,y]D$ when $d(w)$, $d(x)$ and $d(y)$ are missing. This works because

$$M[w,x,y]D = M[w,x,y](EA) = (M[w,x,y]E)A = IA = A$$

even though the w th, x th and y th entries ($d(w)$, $d(x)$ and $d(y)$) of D are unknown (the product $M[w,x,y]D$ is independent of them because the w th, x th and y th columns of $M[w,x,y]$ are zero). It is easy to verify that, in the arithmetic of $GF(4)$, every one of the twenty matrices $M[w,x,y]$ is a left inverse of E . Finally, it is a straightforward matter to produce k out of $k+3$ generalizations of this hyperfast $p/s/r$ process, using only the arithmetic of $GF(4)$.

The major part of the work plan is to write, to run, and to analyze the output of, computer programs for using the heuristic principle to find the encode and decode algorithms for successively larger cases of hyperfast $p/s/r$ processes. This will involve a great deal of run time. Hence it will be necessary to obtain an IBM PC and

use it throughout the project. See Section 8 below. First we propose to find the form of general:

k out of $k+4$; k out of $k+5$; and k out of $k+6$

p/s/r processes using only GF(8) arithmetic, then general

k out of $k+7$, ..., k out of $k+14$

p/s/r processes using only GF(16) arithmetic, then general

k out of $k+15$, ..., k out of $k+30$

p/s/r processes using only GF(32) arithmetic, and so on. These results, which already contain the larger part of the foreseeable practical use of cheap hyperfast p/s/r processes, can be expected to lead to the form of the general k out of $k+j$

p/s/r process using only the arithmetic of GF(2^b), where $j < 2^b$.

Once this is done, the rest of the work plan is to do an abstract design of hardware implementation of p/s/r processes. For packet switching [SL81] and other sequential-arrival-of-words type applications, low cost and minimum parallelism may be the overriding design consideration. For other applications, perhaps involving physically parallel channels transmitting concurrently, cost and use of off-the-shelf components may take a back seat to speed. In this case it may be necessary to provide abstract designs of encode and decode processes utilizing parallel processing, or even the ultimate ultraparallel implementation so as to approach the four-machine-cycle ideal of encoding and decoding speed mentioned in Section 5 above.

The last part of the work plan, also an abstract design task, is to sacrifice speed or economy or both so as to produce general purpose decoders. In other words we want to classify the pairs $((k,n), (k^*,n^*))$ with the property that an encoder (resp. decoder) for a k out of n p/s/r process will encode (resp. decode) for a k^* out of n^* process as well.

Cases of this are known. It is easy to turn the implementation of a 3 out of 6 hyperfast p/s/r process using only GF(4) arithmetic into the implementation of a 2 out of 4 hyperfast p/s/r process using only GF(4) arithmetic by "tying some channels to ground", i.e. by sending only zeros over them (or having the receiver pretend that only zeros are sent over them). We omit details, which a reader can easily work out. Obviously you pay a price in bandwidth. In this example a 3 megabit per second throughput is reduced to 2 megabits per second. It is reasonable to conjecture that a k out of n implementation can be trivially turned into a k^* out of n^* implementation this way if $k^* \leq k$, $n^* \leq n$, and $n^* - k^* \leq n - k$. It would be desirable to verify this conjecture and, if possible, extend it. The advantage of having a few versatile boxes (general purpose communication tools) can sometimes outweigh the panoply of unique advantages peculiar to each of a large number of dedicated boxes (precision single purpose tools) in a military context.

Actual hardware design is not part of Phase I. It will be left to Phase II.

7. Phase I statement of work.

The work will start with the production, and numerous runs, of a program to implement the heuristic device described in Section 6 above. It is strongly indicated by much evidence in the cases $n = k$, $n = k+1$, $n = k+2$, $n = k+3$, and $n = k+7$ that a properly chosen Bloom p/s/r gives rise to an appropriate hyperfast p/s/r for any choice of k and n . The program will produce the list of matrices which embody this hyperfast k out of n case, for each choice of k and n . By the end of two months the first of these results (the cases $4 \leq k \leq 7$, n around 60) will be available. Within the following month or two, the other cases most important to the general solution of the problem of building all hyperfast p/s/r processes should be

available. They may not be in the best form. If not, an interactive matrix manipulation program will be produced to format them in the manner most conducive to reading off the general structure of the matrices which embody a hyperfast p/s/r process. The last two months will be devoted to discovering, and then proving correct, the form of the general hyperfast p/s/r process. Even if the general solution is not found, most cases with any conceivable practical importance will have been settled.

The abstract design principles for implementation can proceed concurrently with the discovery process over the last 3 of the 6 months of the project. The reason for this is that the general form of the solution is known. Both encode and decode are dot products between vectors in an n dimensional or a k dimensional vector space. What is not yet conclusively demonstrated, though we gave a well motivated conjecture in Section 6, is the size of the fields underlying these vector spaces for a given choice of n and k . And the number of occurrences of each member of that field is quite mysterious. But, as these pieces fall into place case by case, the abstract design principles can evolve iteratively.

At the end of the sixth month YLYK Ltd. will deliver a report. The report will contain a catalog of k out of n cases of hyperfast p/s/r processes embodied in lists of matrices for various important values of k and n . If the work meets with complete success it will in fact give the form of the list of matrices embodying the general k out of n hyperfast p/s/r. Finally, it will describe the abstract design principles of implementing such p/s/r processes on currently available off-the-shelf hardware, as well as on a hypothetical vector machine or even a hypothetical ultraparallel processor.

8. Facilities/equipment.

So far the hyperfast k out of $k+1$, 4 out of 6, 3 out of 6, and 7 out of 14 cases of a p/s/r process have been produced with no more computer power than an HP 41C. This is because the fields in question are quite small. Hence no matrix larger than 14 by 14 is needed to turn the heuristic principle described in Section 6 above into an infinite collection of encode/decode rules. But in order to go beyond this it will be necessary to at least double the size of the Galois field F in question. It will also be necessary to do linear algebra with matrices larger than 30 by 30. And by the time the general form of the encode/decode procedure for k out of n processes emerges we will probably be dealing with something like a k out of $k+100$ case. This will involve fields with more than 100 elements, and (extremely sparse) matrices of size approximately 20,000 by 20,000 over such fields.

The calculations involved will require a computer capable of supporting FORTRAN, as well as being easily programmable in its own assembly language, and with sizable memory. The IBM Personal Computer is just about the smallest of the machines capable of carrying out this program. But with 64K RAM, and assuming adroit programming and use of disk memory, it will be possible to explore the consequences of the heuristic principle mentioned in Section 6 above within the size ranges aforementioned. No other special equipment will be required to complete the project.

The 2-room facilities available to YLYK Ltd. at Ann Arbor are adequate to the task at hand. They can accommodate the IBM PC and provide the principal investigator with a work area and necessary library and drafting facilities. Other personnel can be accommodated there, or else assigned duties to be performed on their own premises in consultant fashion.

9. Consultants.

Charles Asmuth (Ph.D., Mathematics, University of Chicago, 1976) did postdoctoral work at the Institute for Advanced Study in Princeton, New Jersey. He taught in the

department of mathematics at Texas A&M before taking his present position as assistant professor in the department of mathematics and computer science at Rutgers University (Newark). He is author or coauthor of some ten papers in mathematics and its applications, especially information theory and cryptography. He will be a consultant on the proposed research. His combination of knowledge in electrical engineering, computer science and abstract algebra will be useful in going from the classification of hyperfast p/s/r processes to implementation.

G. R. Blakley (Ph.D. Mathematics, University of Maryland, 1960) did postdoctoral work at Cornell and Harvard. He has been on the mathematics department faculty of the University of Illinois (Urbana), SUNY at Buffalo, and Texas A&M (where he was department head for many years, and where he is currently a professor). He is author or coauthor of some 30 papers in mathematics and its applications, especially information theory and cryptography. He will be a consultant on the proposed research. His expertise in linear algebra will be useful in finding a general scheme under which the anticipated abundance of hyperfast p/s/r processes can be classified.

John Bloom (Ph.D., Mathematics, CalTech, 1977) taught at the department of mathematics, Texas A&M University, before taking his present research and development position at Chevron, La Habra, California. He is author or coauthor of some ten papers and technical reports in mathematics and its applications, including information theory. He will be a consultant on the proposed research. His expertise in algebraic number theory and algebraic geometry will be especially useful in the very first phase, formulating the programs which implement the heuristic based on the Bloom p/s/r processes and produce examples of hyperfast p/s/r processes for various choices of k and n .

10. Related work. Bibliographic citations list.

Bob Blakley served as a draftsman for the City of Bryan, Texas, in the summer of 1978. He is an expert scientific programmer, having been employed at various times over the last three years in software production and maintenance by research contracts and grants in the Mathematics, Mechanical Engineering, Statistics, Chemistry, Biochemistry and Biophysics departments of Texas A&M University, the Geophysical Fluid Dynamics Laboratory at Princeton University and the University of Michigan Computer Center, as well as for YLYK Ltd. of Ann Arbor, Michigan. He has had extensive experience in algebraic scientific software production, some of it in collaboration with G. R. Blakley. He has produced sizable module [HE74] theoretic generalizations of linear algebraic programs for chemical applications. He has produced programs for the arithmetic of g -adic rings [MA81] and the arithmetic of finite fields of characteristic 2. He has implemented computer simulations of both the Asmuth-Blakley [AS82] p/s/r analog of the Asmuth-Bloom threshold scheme [AS83] and the Bloom-style p/s/r analog of the Bloom threshold scheme [BL81b]. He has a substantial academic background in logic, computer science and natural languages. He is conversant with a dozen computer languages, several of which are assembly languages.

C. A. Asmuth is one of the leading practitioners in the theory of threshold schemes [AS83], p/s/r processes [AS82] and their applications [AS81]. He has a practical familiarity with digital electronics extending back many years. His grasp of abstract algebra and abstract harmonic analysis is highly sophisticated.

G. R. Blakley invented [BL79] threshold schemes, and is a major contributor [BL80; BL81a; BL82] to their theory. With Asmuth, he first explicitly identified the red noise problem [AS82] and solved it (though Bloom certainly [BL81b] foreshadowed this solution). He works actively [BL83b] on minimal computational complexity algorithms for scientific and mathematical computations. His interest in linear algebra, and its applications outside mathematics, goes back twenty years, and has issued in numerous publications not cited here because they are not directly relevant.

to the topic at hand. The term linear algebra is used here in an expansive sense which includes matrix analysis on the analytic side, and integer matrices -- and, more generally, module theory -- on the abstract algebraic side. He is currently principal investigator on a National Security Agency grant to do unclassified research in information theory, some aspects of which are related to the theory and practice of p/s/r processes.

J. Bloom is the inventor of the Bloom threshold scheme [BL81b], the fastest known. His work prefigured the development of the Bloom-style p/s/r processes and the hyperfast p/s/r processes. His influence is major and his insight into every aspect of the subject is incisive. His grasp of geometry, including algebraic geometry, is powerful. He has devoted the last two years to sophisticated programming efforts on computers near the edge of the envelope.

C. Asmuth, Bob Blakley, G. R. Blakley and J. Bloom have all known each other for more than five years. They communicate effortlessly with each other on technical matters. The requested travel funds will be used to get two or more of them together for periods of several days at several points during the work.

BIBLIOGRAPHIC CITATIONS LIST

- AB83 P. H. Abelson, Glass fiber communication, Science, Vol. 220 (1983), p. 463.
- AN83 Anonymous, C³ experiment explores data restoration, Aviation Week and Space Technology, Vol. 118, no. 17, April 25, (1983), pp. 155-158.
- AS81 C. A. Asmuth and G. R. Blakley, An efficient algorithm for constructing a cryptosystem which is harder to break than two other cryptosystems, Computers and Mathematics with Applications, Vol. 7 (1981), p. 447-449.
- AS82 C. A. Asmuth and G. R. Blakley, Pooling, splitting and restituting information to overcome total failure of channels of communication, Proceedings of the 1982 Symposium on Security and Privacy, IEEE Computer Society, Los Angeles, California (1982), pp. 156-169.
- AS83 C. A. Asmuth and J. Bloom, A modular approach to key safeguarding, IEEE Transactions on Information Theory, Vol. IT-30 (1983), pp. 208-210.
- BE68 E. R. Berlekamp, Algebraic Coding Theory, McGraw-Hill, New York (1968).
- BE74 E. R. Berlekamp (Editor), Key Papers in the Development of Coding Theory, IEEE Press, New York (1974).
- BE78 E. R. Berlekamp, R. J. McEliece and H. G. A. van Tilborg, On the inherent intractability of certain coding problems, IEEE Transactions on Information Theory, Vol. IT-24 (1978), pp. 384-386.
- BL79 G. R. Blakley, Safeguarding cryptographic keys, Proceedings of the National Computer Conference, 1979, AFIPS Conference Proceedings, Vol. 48 (1979), pp. 313-317.
- BL80 G. R. Blakley, One-time pads are key safeguarding schemes, not cryptosystems. Fast key safeguarding schemes (threshold schemes) exist, Proceedings of the 1980 Symposium on Security and Privacy, IEEE Computer Society, New York (1980), pp. 108-113.
- BL81a G. R. Blakley and Laif Swanson, Security proofs for information protection systems, Proceedings of the 1981 Symposium on Security and Privacy, IEEE Computer Society, Los Angeles (1981), pp. 75-88.
- BL81b J. Bloom, A note on superfast threshold schemes, Preprint, Texas A&M University, Department of Mathematics (1981), and Threshold schemes and error correcting codes, Abstracts of Papers Presented to the American Mathematical Society, Vol. 2 (1981), p. 230.
- BL82 G. R. Blakley, Protecting information against both destruction and unauthorized disclosure, Proceedings of the 1982 Carnahan Conference on Security Technology, Univ. of Kentucky Press (1982), pp. 123-133.
- BL83a R. E. Blahut, Theory and Practice of Error Control Codes, Addison-Wesley, Reading, Massachusetts (1983).

- BL83b G. R. Blakley, A computer algorithm for calculating the product AB modulo M , IEEE Transactions on Computers, Vol. C-32 (1983), in press.
- BR81 W. J. Broad, Reagan eyes the message gap, Science, Vol. 214 (1981), p. 312.
- DA80 G. I. Davida, R. A. DeMillo and R. J. Lipton, Protecting shared cryptographic keys, Proceedings of the 1980 Symposium on Security and Privacy, IEEE Computer Society, New York (1980), pp. 100-102.
- DE82 D. Denning, Cryptography and Data Security, Addison-Wesley, Reading, Massachusetts (1982).
- DI79 W. Diffie and M. Hellman, Privacy and authentication: An introduction to cryptography, Proceedings of the IEEE, Vol. 67 (1979), pp. 397-427.
- EL83 B. M. Elson, USAF studies new computer concept, Aviation Week and Space Technology, Vol. 118, no. 19, 9 May (1983), pp. 69-71.
- GA78 M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco (1978).
- GA83 J. Garrett, Pulse-position modulation for transmission over optical fibers with direct or heterodyne detection, IEEE Transactions on Communications, Vol. COM 31 (1983), pp. 518-527.
- G083 R. J. Godin, Laser tool brings ultrafast devices closer, Electronics, Vol. 56, no. 9, 5 May (1983), pp. 112-114.
- HA80 R. W. Hamming, Coding and Information Theory, Prentice-Hall, Englewood Cliffs, New Jersey (1980).
- HE74 T. Head, Modules: A Primer of Structure Theorems, Brooks Cole, Monterey, California (1974).
- H071 K. Hoffman and R. Kunze, Linear Algebra, Second Edition, Prentice Hall, Englewood Cliffs, New Jersey (1971).
- JA64 N. Jacobson, Lectures in Abstract Algebra, Volume 3, Theory of Fields and Galois Theory, D. Van Nostrand, Princeton, New Jersey.
- KA81 E. D. Karnin, J. W. Greene and M. E. Hellman, On secret sharing systems, Verbal presentation, Session B3 (Cryptography), 1981 IEEE International Symposium on Information Theory, Santa Monica, California, February 9-12 (1981) and On secret sharing systems, Preprint, Stanford University, Department of Electrical Engineering (1981).
- K081 A. G. Konheim, Cryptography: A Primer, Wiley-Interscience, New York (1981).
- LI83 T. Li, Advances in optical fiber communications: An historical perspective, IEEE Journal on Selected Areas in Communications, Vol. SAC-1 (1983), pp. 356-372.
- MA67 S. MacLane and G. Birkhoff, Algebra, MacMillan, New York, 1967.
- MA77 F. J. MacWilliams and N. J. A. Sloane, The Theory of Error Correcting Codes, North-Holland, Amsterdam (1978).
- MA81 K. Mahler, p -adic Numbers and Their Functions, Second Edition, Cambridge University Press (1981).
- MC77 R. J. McEliece, The Theory of Information and Coding, Addison-Wesley, Reading, Massachusetts (1977).
- MC81 R. J. McEliece and D. V. Sarwate, On sharing secrets and Reed-Solomon codes, Communications of the ACM, Vol. 24 (1981), pp. 583-584.
- NE83 S. B. Newell, A. J. de Geus and R. A. Rohrer, Design automation for integrated circuits, Science, Vol. 220 (1983), pp. 465-472.
- N069 B. Noble, Applied Linear Algebra, Prentice-Hall, Englewood Cliffs, New Jersey (1969).
- NT81 S. C. Ntafos and S. L. Hakimi, On the complexity of some coding problems, IEEE Transactions on Information Theory, Vol. IT-27 (1981), pp. 794-796.
- PE72 W. W. Peterson and E. J. Weldon, Jr., Error Correcting Codes, Second Edition, M.I.T. Press, Cambridge, Massachusetts (1972).
- PL82 V. Pless, Introduction to the Theory of Error Correcting Codes, Wiley-Interscience, New York (1982).

- PO82 R. D. Posner, Packet Switching, Lifetime Learning Publications, Belmont, California (1982).
- SH49 C. E. Shannon, Communication theory of secrecy systems, Bell System Technical Journal, Vol. 28 (1949), pp. 656-715.
- RA83 S. C. Rashleigh and R. H. Stolen, Preservation of polarization in single mode fibers, Laser Focus with Fiberoptic Technology, Vol. 19, no. 5, May (1983), pp. 155-161.
- RE60 I. S. Reed and G. Solomon, Polynomial codes over certain finite fields, J. SIAM, Vol. 8 (1960), pp. 300-304.
- RE67 L. Redei, Algebra, Volume 1, Pergamon Press, Oxford (1967).
- RO83 M. J. Ross, Military/government digital switching systems, IEEE Communications Magazine, Vol. 21, no. 3, May (1983) pp. 18-25.
- SH79 A. Shamir, How to share a secret, Communications of the ACM, Vol. 22 (1979), pp. 612-613.
- SL81 M. F. Slana and H. R. Lehman, Data communication using the telecommunication network, Computer, Vol. 14, no. 5, May (1981), pp. 73-88.
- ST63 J. J. Stone, Multiple-burst error correction with the Chinese remainder theorem, J. SIAM, Vol. 11 (1963), pp. 74-81.
- ST83 B. W. Stuck, Calculating the maximum mean data rate in local area networks, Computer, Vol. 16, No. 5, May (1983), pp. 72-76.
- TH83 D. E. Thomsen, A pure laser for clean communications, Science News, Vol. 123 (1983), p. 260.
- VI79 A. J. Viterbi and J. K. Omura, Principles of Digital Communication and Coding, McGraw-Hill, New York (1979).
- YA68 P. B. Yale, Geometry and Symmetry, Holden-Day, San Francisco (1968).

11. Key Personnel.

YLYK Ltd. was incorporated in Delaware on 4 June 1979. It is currently headquartered in Ann Arbor, Michigan. It has produced software, designed algorithms, designed systems in the area of coding, communications and cryptography, and has conducted studies.

Bob Blakley, born 13 July 1960 in Washington D.C., is a citizen of the U.S.A. and a 1982 honors graduate of Princeton University. He married Karen Hejtmancik of College Station, Texas, on 7 August 1982. His previous technical employment history can be found in Section 10 above. He is currently involved in part time teaching and graduate study in computer science at the University of Michigan. He is coauthor of three papers on cryptography and information theory in Cryptologia, Volume 2 (1978), pp. 305-321, Volume 3 (1979), pp. 29-42, and Volume 3 (1979), pp. 105-118. He is president of YLYK Ltd., and will be principal investigator on the proposed research. His Social Security Number is 460-06-2353.

12. Current and pending support.

SBIR proposals very similar to this proposal, all bearing the title

High-speed low-cost ways to get messages from a sender to a receiver when some channels linking them become inoperative,

and all having Bob Blakley, President, YLYK Ltd., as principal investigator are being submitted in May 1983 to the following DOD components under DOD Program Solicitation Number 83.1, Small Business Research Program, Closing date 31 May 1983:

Appendix B

Tables of $GF(2^N)$ arithmetic

Addition Table for GF $2^{**}(2)$ Mod(7)

+	1	0	1	2	3
0	1	0	1	2	3
1	1	1	0	3	2
2	1	2	3	0	1
3	1	3	2	1	0

Multiplication Table for GF $2^{**}(2)$ Mod(7)

+	1	0	1	2	3
0	1	0	0	0	0
1	1	0	1	2	3
2	1	0	2	3	1
3	1	0	3	1	2

Addition Table for GF $2^{**}(3)$ Mod(13)

+	1	0	1	2	3	4	5	6	7
0	1	0	1	2	3	4	5	6	7
1	1	1	0	3	2	5	4	7	6
2	1	2	3	0	1	6	7	4	5
3	1	3	2	1	0	7	6	5	4
4	1	4	5	6	7	0	1	2	3
5	1	5	4	7	6	1	0	3	2
6	1	6	7	4	5	2	3	0	1
7	1	7	6	5	4	3	2	1	0

Multiplication Table for GF $2^{**}(3)$ Mod(13)

+	1	0	1	2	3	4	5	6	7
0	1	0	0	0	0	0	0	0	0
1	1	0	1	2	3	4	5	6	7
2	1	0	2	4	6	3	1	7	5
3	1	0	3	6	5	7	4	1	2
4	1	0	4	3	7	6	2	5	1
5	1	0	5	1	4	2	7	3	6
6	1	0	6	7	1	5	3	2	4
7	1	0	7	5	2	1	6	4	3

Addition Table for $GF(2^{**}(4) \text{ Mod}(23))$

+	1	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
00	1	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
01	1	01	00	03	02	05	04	07	06	11	10	13	12	15	14	17	16
02	1	02	03	00	01	06	07	04	05	12	13	10	11	16	17	14	15
03	1	03	02	01	00	07	06	05	04	13	12	11	10	17	16	15	14
04	1	04	05	06	07	00	01	02	03	14	15	16	17	10	11	12	13
05	1	05	04	07	06	01	00	03	02	15	14	17	16	11	10	13	12
06	1	06	07	04	05	02	03	00	01	16	17	14	15	12	13	10	11
07	1	07	06	05	04	03	02	01	00	17	16	15	14	13	12	11	10
10	1	10	11	12	13	14	15	16	17	00	01	02	03	04	05	06	07
11	1	11	10	13	12	15	14	17	16	01	00	03	02	05	04	07	06
12	1	12	13	10	11	16	17	14	15	02	03	00	01	06	07	04	05
13	1	13	12	11	10	17	16	15	14	03	02	01	00	07	06	05	04
14	1	14	15	16	17	10	11	12	13	04	05	06	07	00	01	02	03
15	1	15	14	17	16	11	10	13	12	05	04	07	06	01	00	03	02
16	1	16	17	14	15	12	13	10	11	06	07	04	05	02	03	00	01
17	1	17	16	15	14	13	12	11	10	07	06	05	04	03	02	01	00

Multiplication Table for $GF(2^{**}(4) \text{ Mod}(23))$

+	1	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
00	1	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
01	1	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
02	1	00	02	04	06	10	12	14	16	03	01	07	05	13	11	17	15
03	1	00	03	06	05	14	17	12	11	13	10	15	16	07	04	01	02
04	1	00	04	10	14	03	07	13	17	06	02	16	12	05	01	15	11
05	1	00	05	12	17	07	02	15	10	16	13	04	01	11	14	03	06
06	1	00	06	14	12	13	15	07	01	05	03	11	17	16	10	02	04
07	1	00	07	16	11	17	10	01	06	15	12	03	04	02	05	14	13
10	1	00	10	03	13	06	16	05	15	14	04	17	07	12	02	11	01
11	1	00	11	01	10	02	13	03	12	04	15	05	14	06	17	07	16
12	1	00	12	07	15	16	04	11	03	17	05	10	02	01	13	06	14
13	1	00	13	05	16	12	01	17	04	07	14	02	11	15	06	10	03
14	1	00	14	13	07	05	11	16	02	12	06	01	15	17	03	04	10
15	1	00	15	11	04	01	14	10	05	02	17	13	06	03	16	12	07
16	1	00	16	17	01	15	03	02	14	11	07	06	10	04	12	13	05
17	1	00	17	15	02	11	06	04	13	01	16	14	03	10	07	05	12

Addition Table for GF 2**(5) Mod(45)

+	1	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17	20	21	22	23	24	25	26	27	30	31	32	33	34	35	36	37
00	1	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17	20	21	22	23	24	25	26	27	30	31	32	33	34	35	36	37
01	1	01	00	03	02	05	04	07	06	11	10	13	12	15	14	17	16	21	20	23	22	25	24	27	26	31	30	33	32	35	34	37	36
02	1	02	03	00	01	06	07	04	05	12	13	10	11	16	17	14	15	22	23	20	21	26	27	24	25	32	33	30	31	36	37	34	35
03	1	03	02	01	00	07	06	05	04	13	12	11	10	17	16	15	14	23	22	21	20	27	26	25	24	33	32	31	30	37	36	35	34
04	1	04	05	06	07	00	01	02	03	14	15	16	17	10	11	12	13	24	25	26	27	20	21	22	23	34	35	36	37	30	31	32	33
05	1	05	04	07	06	01	00	03	02	15	14	17	16	11	10	13	12	25	24	27	26	21	20	23	22	35	34	37	36	31	30	33	32
06	1	06	07	04	05	02	03	00	01	16	17	14	15	12	13	10	11	26	27	24	25	22	23	20	21	36	37	34	35	32	33	30	31
07	1	07	06	05	04	03	02	01	00	17	16	15	14	13	12	11	10	27	26	25	24	23	22	21	20	37	36	35	34	33	32	31	30
10	1	10	11	12	13	14	15	16	17	00	01	02	03	04	05	06	07	30	31	32	33	34	35	36	37	20	21	22	23	24	25	26	27
11	1	11	10	13	12	15	14	17	16	01	00	03	02	05	04	07	06	31	30	33	32	35	34	37	36	21	20	23	22	25	24	27	26
12	1	12	13	10	11	16	17	14	15	02	03	00	01	06	07	04	05	32	33	30	31	36	37	34	35	22	23	20	21	26	27	24	25
13	1	13	12	11	10	17	16	15	14	03	02	01	00	07	06	05	04	33	32	31	30	37	36	35	34	23	22	21	20	27	26	25	24
14	1	14	15	16	17	10	11	12	13	04	05	06	07	00	01	02	03	34	35	36	37	30	31	32	33	24	25	26	27	20	21	22	23
15	1	15	14	17	16	11	10	13	12	05	04	07	06	01	00	03	02	35	34	37	36	31	30	33	32	25	24	27	26	21	20	23	22
16	1	16	17	14	15	12	13	10	11	06	07	04	05	02	03	00	01	36	37	34	35	32	33	30	31	26	27	24	25	22	23	20	21
17	1	17	16	15	14	13	12	11	10	07	06	05	04	03	02	01	00	37	36	35	34	33	32	31	30	27	26	25	24	23	22	21	20
20	1	20	21	22	23	24	25	26	27	30	31	32	33	34	35	36	37	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
21	1	21	20	23	22	25	24	27	26	31	30	33	32	35	34	37	36	01	00	03	02	05	04	07	06	11	10	13	12	15	14	17	16
22	1	22	23	20	21	26	27	24	25	32	33	30	31	36	37	34	35	02	03	00	01	06	07	04	05	12	13	10	11	16	17	14	15
23	1	23	22	21	20	27	26	25	24	33	32	31	30	37	36	35	34	03	02	01	00	07	06	05	04	13	12	11	10	17	16	15	14
24	1	24	25	26	27	20	21	22	23	34	35	36	37	30	31	32	33	04	05	06	07	00	01	02	03	14	15	16	17	10	11	12	13
25	1	25	24	27	26	21	20	23	22	35	34	37	36	31	30	33	32	05	04	07	06	01	00	03	02	15	14	17	16	11	10	13	12
26	1	26	27	24	25	22	23	20	21	36	37	34	35	32	33	30	31	06	07	04	05	02	03	00	01	16	17	14	15	12	13	10	11
27	1	27	26	25	24	23	22	21	20	37	36	35	34	33	32	31	30	07	06	05	04	03	02	01	00	17	16	15	14	13	12	11	10
30	1	30	31	32	33	34	35	36	37	20	21	22	23	24	25	26	27	10	11	12	13	14	15	16	17	00	01	02	03	04	05	06	07
31	1	31	30	33	32	35	34	37	36	21	20	23	22	25	24	27	26	11	10	13	12	15	14	17	16	01	00	03	02	05	04	07	06
32	1	32	33	30	31	36	37	34	35	22	23	20	21	26	27	24	25	12	13	10	11	16	17	14	15	02	03	00	01	06	07	04	05
33	1	33	32	31	30	37	36	35	34	23	22	21	20	27	26	25	24	13	12	11	10	17	16	15	14	03	02	01	00	07	06	05	04
34	1	34	35	36	37	30	31	32	33	24	25	26	27	20	21	22	23	14	15	16	17	10	11	12	13	04	05	06	07	00	01	02	03
35	1	35	34	37	36	31	30	33	32	25	24	27	26	21	20	23	22	15	14	17	16	11	10	13	12	05	04	07	06	01	00	03	02
36	1	36	37	34	35	32	33	30	31	26	27	24	25	22	23	20	21	16	17	14	15	12	13	10	11	06	07	04	05	02	03	00	01
37	1	37	36	35	34	33	32	31	30	27	26	25	24	23	22	21	20	17	16	15	14	13	12	11	10	07	06	05	04	03	02	01	00

Appendix C

Selected tables of Vandermonde matrices

Vandermonde Matrix for GF $2^{+}(2)$ mod 7 is:

```
1 0 0 0
1 1 1 1
1 2 3 4
1 4 2 1
```

Vandermonde Matrix for GF $2^{+}(3)$ mod 13 is:

```
1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1
1 2 4 7 6 7 5 1
1 4 6 5 2 3 7 1
1 7 5 4 7 2 6 1
1 6 2 7 4 5 3 1
1 7 3 2 5 6 4 1
1 5 7 6 3 4 2 1
```

Vandermonde Matrix for GF $2^{+}(4)$ mod 23 is:

```
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 02 04 10 03 06 14 13 05 12 07 16 17 15 11 01
01 04 03 14 05 07 17 11 02 10 06 13 12 16 15 01
01 10 14 12 17 01 10 14 12 17 01 10 14 12 17 01
01 03 05 17 02 06 12 15 04 14 07 11 10 13 16 01
01 06 07 01 06 07 01 06 07 01 06 07 01 06 07 01
01 14 17 10 12 01 14 17 10 12 01 14 17 10 12 01
01 17 11 14 15 06 17 03 16 19 07 04 12 02 05 01
01 05 02 12 04 07 10 16 03 17 06 15 14 11 13 01
01 12 10 17 14 01 12 10 17 14 01 12 10 17 14 01
01 07 06 01 07 06 01 07 06 01 07 06 01 07 06 01
01 16 13 10 11 07 14 04 15 12 06 02 17 05 03 01
01 17 12 14 10 01 17 12 14 10 01 17 12 14 10 01
01 15 16 12 13 06 10 02 11 17 07 05 14 03 04 01
01 11 15 17 16 07 12 05 13 14 06 03 10 04 02 01
```

[illegible]

Appendix D

Tables of ENF (encode normal forms) produced by
cold precomputations

ENF Matrix for GF $2^{*}(2)$ mod 7 is:

```
1 1 1 1
1 2 1 0
```

ENF Matrix for GF $2^{*}(3)$ mod 13 is:

```
1 1 1 1 1 1 1 1
7 2 6 5 3 4 1 0
2 3 2 1 0 1 0 0
3 5 2 5 1 0 0 0
4 3 6 1 0 0 0 0
3 2 1 0 0 0 0 0
```

ENF Matrix for GF $2^{*}(4)$ mod 23 is:

```
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
17 02 06 16 15 13 07 14 11 03 04 12 05 10 01 00
14 10 15 14 04 11 15 04 01 10 05 11 05 01 00 00
07 14 10 05 03 17 03 15 12 03 05 12 01 00 00 00
15 07 02 14 02 04 11 14 12 10 04 01 00 00 00 00
05 01 12 07 15 03 11 15 01 03 01 00 00 00 00 00
02 14 16 06 15 04 03 04 11 01 00 00 00 00 00 00
11 15 15 07 02 17 15 14 01 00 00 00 00 00 00 00
16 14 12 14 03 11 07 01 00 00 00 00 00 00 00 00
13 01 02 06 04 13 01 00 00 00 00 00 00 00 00 00
17 07 10 14 15 01 00 00 00 00 00 00 00 00 00 00
15 14 15 15 01 00 00 00 00 00 00 00 00 00 00 00
17 10 05 01 10 00 00 00 00 00 00 00 00 00 00 00
07 02 01 00 00 00 00 00 00 00 00 00 00 00 00 00
```


Appendix E

Examples of the encode/decode process

Please enter, on one line and separated by a blank,
the field-base and modulus to be used. The
field-base should be a decimal number and the
modulus should be an octal number.

4 23

Please enter the number of channels to be sent
by the transmitting node. This should be a
decimal number.

10

please enter the number of channels active at
the receivers node; this should be a decimal number.

8

DNF matrix for 8 out of 10
channels over GF $2^{*}(4) \bmod 23$ is:

```
01 00 11 17 16 04 15 11 13 02
10 01 14 02 07 01 10 04 01 04
```

please enter, on one line and separated by blanks,
the numbers of the 8 channels active
at the receiving node. These numbers should be decimal.
3 4 5 6 7 8 9 10

Decoder matrix for the active channels listed above is:

```
01 00 11 17 16 04 15 11 13 02
00 01 14 02 07 01 10 04 01 04
00 00 01 00 00 00 00 00 00 00
00 00 00 01 00 00 00 00 00 00
00 00 00 00 01 00 00 00 00 00
00 00 00 00 00 01 00 00 00 00
00 00 00 00 00 00 01 00 00 00
00 00 00 00 00 00 00 01 00 00
```

please enter, on one line and separated by blanks,
the data received on each of the channels active at
the receivers node. The data should be in the form
of octal numbers, and should be entered in order of
increasing channel number.

12 13 14 15 16 17 07 14

the 8 transmitted cleartext words were
(octal numbers expressed in channel order):

10 11 12 13 14 15 16 17

do you want to decode another 8 words?
(type y or n).

n

Please enter, on one line and separated by a blank,
the field-base and modulus to be used. The
field-base should be a decimal number and the
modulus should be an octal number.

4 23

Please enter the number of channels to be sent
by the transmitting node. This should be a
decimal number.

10

please enter the number of channels active at
the receivers node; this should be a decimal number.

8

DNF matrix for 5 out of 10
channels over GF 2^{16} mod 23 is:

01 00 11 17 16 04 15 11 13 02
00 01 14 02 07 01 10 04 01 04

please enter, on one line and separated by blanks,
the numbers of the 3 channels active
at the receiving node. These numbers should be decimal.
2 3 4 6 7 8 9 10

Decoder matrix for the active channels listed above is:

01 02 02 13 00 06 16 01 11 12
00 01 00 00 00 00 00 00 00 00
00 00 01 00 00 00 00 00 00 00
00 00 00 01 00 00 00 00 00 00
00 06 16 14 01 06 05 13 06 13
00 00 00 00 00 01 00 00 00 00
02 00 00 00 00 00 01 00 00 00
00 00 00 00 00 00 00 01 00 00

please enter, on one line and separated by blanks,
the data received on each of the channels active at
the receivers node. The data should be in the form
of octal numbers, and should be entered in order of
increasing channel number.

11 12 13 15 16 17 07 14

the 8 transmitted cleartext words were
(octal numbers expressed in channel order):

10 11 12 13 14 15 16 17

do you want to decode another 8 words?
(type y or n).

n

Please enter, on one line and separated by a blank,
the field-base and modulus to be used. The
field-base should be a decimal number and the
modulus should be an octal number.

4 23

Please enter the number of channels to be sent
by the transmitting node. This should be a
decimal number.

10

please enter the number of channels active at
the receiver's node; this should be a decimal number.

8

ENF matrix for 8 out of 10
channels over GF $2^{11}(1) \bmod 23$ is:

```
01 00 11 17 16 04 15 11 13 02
00 01 14 02 07 01 10 04 01 04
```

please enter, on one line and separated by blanks,
the numbers of the 8 channels active
at the receiver node. These numbers should be decimal.

2 3 4 5 6 7 8 10

Decoder matrix for the active channels listed above is:

```
01 13 04 12 12 17 12 03 00 10
00 01 00 00 00 00 00 00 00 00
10 00 01 00 00 00 00 00 00 00
00 00 00 01 00 00 00 00 00 00
00 00 00 00 01 00 00 00 00 00
00 00 00 00 00 01 00 00 00 00
00 00 00 00 00 00 01 00 00 00
00 00 00 00 00 00 00 01 00 00
```

please enter, on one line and separated by blanks,
the data received on each of the channels active at
the receiver's node. The data should be in the form
of octal numbers, and should be entered in order of
increasing channel number.

11 12 13 14 15 16 17 14

the 8 transmitted cleartext words were
(octal numbers expressed in channel order):

10 11 12 13 14 15 16 17

do you want to decode another 8 words?
(type y or n).

n

Please enter, on one line and separated by a blank,
the field-base and modulus to be used. The
field-base should be a decimal number and the
modulus should be an octal number.

4 23

Please enter the number of channels to be sent
by the transmitting node. This should be a
decimal number.

10

please enter the number of channels active at
the receiver's node; this should be a decimal number.

8

BNF matrix for 3 out of 10
channels over GF $2^{**}4$ mod 23 is:

E5

01 00 11 17 16 04 15 11 13 02
00 01 14 02 07 01 10 04 01 04

please enter, on one line and separated by blanks,
the numbers of the 8 channels active
at the receiving node. These numbers should be decimal.
1 2 3 4 5 6 7 8 10

Decoder matrix for the active channels listed above is:

01 00 00 00 00 00 00 00 00 00
00 01 00 00 00 00 00 00 00 00
00 00 01 00 00 00 00 00 00 00
00 00 00 01 00 00 00 00 00 00
14 15 05 01 01 10 01 07 00 12
00 00 00 00 00 01 00 00 00 00
00 00 00 00 00 00 01 00 00 00
01 10 00 00 00 00 00 01 00 00

please enter, on one line and separated by blanks,
the data received on each of the channels active at
the receivers node. The data should be in the form
of octal numbers, and should be entered in order of
increasing channel number.
10 11 12 13 15 16 17 14

the 3 transmitted cleartext words were
(octal numbers expressed in channel order):

10 11 12 13 14 15 16 17

do you want to decode another 8 words?
(type y or n).

n
Please enter, on one line and separated by a blank,
the field-base and modulus to be used. The
field-base should be a decimal number and the
modulus should be an octal number.

4 23

Please enter the number of channels to be sent
by the transmitting node. This should be a
decimal number.

10

Please enter the number of channels
the receivers node; this should be a decimal number.
8

BNF matrix for 3 out of 10
channels over GF $2^{**}(4)$ mod 23 is:

01 00 11 17 16 04 15 11 13 02
00 01 14 02 07 01 10 04 01 04

please enter, on one line and separated by blanks,
the numbers of the 8 channels active
at the receiving node. These numbers should be decimal.
1 2 3 4 5 6 7 8

Decoder matrix for the active channels listed above is:

01 00 00 00 00 00 00 00 00 00
00 01 00 00 00 00 00 00 00 00
00 00 01 00 00 00 00 00 00 00

00 00 00 00 01 00 00 00 00 00
 00 00 00 00 00 01 00 00 00 00
 00 00 00 00 00 00 01 00 00 00
 00 00 00 00 00 00 00 01 00 00

please enter, on one line and separated by blanks,
 the data received on each of the channels active at
 the receivers node. The data should be in the form
 of octal numbers, and should be entered in order of
 increasing channel number.
 10 11 12 13 14 15 16 17

the 8 transmitted cleartext words were
 (octal numbers expressed in channel order):

10 11 12 13 14 15 16 17
 do you want to decode another 8 words?
 (type y or n).
 n

please enter, on one line and separated by blanks, the field-base, modulus, number of channels to be sent, and number of channels to be received. The modulus should be an octal number; all other numbers should be decimal.

4 23 10 3

thank you...please wait

ENF MATRIX-----

[illegible]

SUBMATRIX----->

02	14	15	06	15	04	03	04	11	01
11	15	16	07	02	17	15	14	01	00
06	14	12	14	03	11	07	01	00	00
13	01	02	06	04	13	01	00	00	00
17	07	10	14	15	01	00	00	00	00
15	14	15	16	01	00	00	00	00	00
17	10	06	01	00	00	00	00	00	00

ENCODE KEY----->

16	11	06	00	00	00	00	00	00	01
13	04	16	00	00	00	00	00	01	00
05	15	11	00	00	00	00	01	00	00
16	10	07	00	00	00	01	00	00	00
01	01	01	00	00	01	00	00	00	00
13	05	17	00	01	00	00	00	00	00
17	10	06	01	00	00	00	00	00	00

please enter, on one line, in octal and separated
by blanks, the values to be transmitted over the
transmitters 3 channels

0 1 2

words transmitted are (in channel order):

00 01 02 04 10 03 06 14 13 05

do you want to send another 3 words?

(type y or n)

y

please enter, on one line, in octal and separated
by blanks, the values to be transmitted over the
transmitters 3 channels

3 4 5

words transmitted are (in channel order):

03 04 05 11 17 02 17 05 16 16

do you want to send another 3 words?

(type y or n)

y

please enter, on one line, in octal and separated
by blanks, the values to be transmitted over the
transmitters 3 channels

6 7 10

words transmitted are (in channel order):

16 07 10 14 06 11 02 14 11 15

do you want to send another 3 words?

(type y or n)

y

please enter, on one line, in octal and separated
by blanks, the values to be transmitted over the
transmitters 3 channels

11 12 13

words transmitted are (in channel order):

11 12 13 16 13 10 14 14 12 15

do you want to send another 3 words?

(type y or n)

y

please enter, on one line, in octal and separated
by blanks, the values to be transmitted over the
transmitters 3 channels

14 15 16

words transmitted are (in channel order):

14 15 16 10 04 17 12 00 07 11

do you want to send another 3 words?

(type y or n)

y

please enter, on one line, in octal and separated
by blanks, the values to be transmitted over the
transmitters 3 channels

17 10 4

words transmitted are (in channel order):

17 10 04 15 04 03 05 05 10 12

do you want to send another 3 words?

(type y or n)

y

please enter, on one line and separated by blanks, the field-base, modulus, number of channels to be sent, and number of channels to be received. The modulus should be an octal number; all other numbers should be decimal.

4 27 10 5

thank you...please wait

DEF MATRIX----->

[illegible]

AD-A142 831 HIGH SPEED LOW-COST WAYS TO GET MESSAGES FROM A SENDER 2/2

TO A RECEIVER WHEN (U) YLYK LTD ANN ARBOR MI
B BLAKLEY 28 MAY 84 YLYK/AFOSR/SBIRI/83-84/001

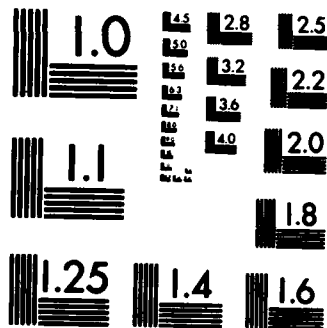
UNCLASSIFIED AFOSR-TR-84-0528 F49620-83-C-0160

F/G 17/2.1 NL

END

FILED

DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SUBMATRIX-----

02	14	16	06	15	04	03	04	11	01
11	15	16	07	02	17	15	14	01	00
06	14	12	14	03	11	07	01	00	00
13	01	02	06	04	13	01	00	00	00
17	07	10	14	15	01	00	00	00	00

ENCODE KEY----->

03	06	03	17	10	00	00	00	00	01
10	16	15	15	07	00	00	00	01	00
05	16	06	16	02	00	00	01	00	00
10	05	05	13	02	00	01	00	00	00
17	07	10	14	15	01	00	00	00	00

please enter, on one line, in octal and separated
by blanks, the values to be transmitted over the
transmitters 5 channels

0 1 2 3 4

words transmitted are (in channel order):

00 01 02 03 04 02 11 13 14 04

do you want to send another 5 words?

(type y or n)

y

please enter, on one line, in octal and separated
by blanks, the values to be transmitted over the
transmitters 5 channels

5 6 7 10 11

words transmitted are (in channel order):

05 06 07 10 11 17 15 11 01 04

do you want to send another 5 words?

(type y or n)

y

please enter, on one line, in octal and separated
by blanks, the values to be transmitted over the
transmitters 5 channels

12 13 14 15 16

words transmitted are (in channel order):

12 13 14 15 16 13 16 07 06 13

do you want to send another 5 words?

(type y or n)

y

please enter, on one line, in octal and separated
by blanks, the values to be transmitted over the
transmitters 5 channels

17 10 4 14 0

words transmitted are (in channel order):

17 10 04 14 00 16 05 00 12 03

do you want to send another 5 words?

(type y or n)

n

please enter, on one line and separated by blanks, the field-base, modulus, number of channels to be sent, and number of channels to be received. The modulus should be an octal number; all other numbers should be decimal.

4 23 10 8

thank you...please wait

ENF MATRIX----->

[illegible]

SUBMATRIX----->

02 14 16 06 15 04 03 04 11 01
11 15 16 07 02 17 15 14 01 00

ENCODE KEY----->

17 03 11 14 14 12 14 02 00 01

11 15 16 07 02 17 15 14 01 00

please enter, on one line, in octal and separated
by blanks, the values to be transmitted over the
transmitters 8 channels

0 1 2 3 4 5 6 7

words transmitted are (in channel order):

00 01 02 03 04 05 06 07 17 04

do you want to send another 8 words?

(type y or n)

y

please enter, on one line, in octal and separated
by blanks, the values to be transmitted over the
transmitters 8 channels

10 11 12 13 14 15 16 17

words transmitted are (in channel order):

10 11 12 13 14 15 16 17 07 14

do you want to send another 8 words?

(type y or n)

n

Please enter, on one line and separated by a blank, the field-base and modulus to be used. The field-base should be a decimal number and the modulus should be an octal number.

4 23

Please enter the number of channels to be sent by the transmitting node. This should be a decimal number.

10

please enter the number of channels active at the receivers node; this should be a decimal number.

3

DNF matrix for 3 out of 10 channels over GF $2^{**}(4)$ mod 23 is:

```
01 00 00 00 00 00 00 14 05 10
00 01 00 00 00 00 00 04 12 17
00 00 01 00 00 00 00 01 01 01
00 00 00 01 00 00 00 10 17 06
00 00 00 00 01 00 00 05 12 16
00 00 00 00 00 01 00 11 16 06
00 00 00 00 00 00 01 05 13 17
```

please enter, on one line and separated by blanks, the numbers of the 3 channels active at the receiving node. These numbers should be decimal.

1 5 8

Decoder matrix for the active channels listed above is:

```
01 00 00 00 00 00 00 00 00 00
10 01 00 00 05 00 00 14 00 00
16 00 01 00 13 00 00 04 00 00
```

please enter, on one line and separated by blanks, the data received on each of the channels active at the receivers node. The data should be in the form of octal numbers, and should be entered in order of increasing channel number.

6 6 14

the 3 transmitted cleartext words were (octal numbers expressed in channel order):

06 07 10

do you want to decode another 3 words? (type y or n).

y

please enter, on one line and separated by blanks, the data received on each of the channels active at the receivers node. The data should be in the form of octal numbers, and should be entered in order of increasing channel number.

3 17 5

the 3 transmitted cleartext words were (octal numbers expressed in channel order):

03 04 05

do you want to decode another 3 words?

0
Please enter, on one line and separated by a blank,
the field-base and modulus to be used. The
field-base should be a decimal number and the
modulus should be an octal number.

4 23

Please enter the number of channels to be sent
by the transmitting node. This should be a
decimal number.

10

Please enter the number of channels active at
the receivers node; this should be a decimal number.

5

ENF matrix for 5 out of 10
channels over GF $2^{**}(4)$ mod 23 is:

```
01 00 00 00 00 15 14 12 02 10
00 01 00 00 00 14 14 13 03 11
10 00 01 00 00 12 02 16 02 05
00 00 00 01 00 10 14 05 13 13
00 00 00 00 01 04 13 06 16 06
```

Please enter, on one line and separated by blanks,
the numbers of the 5 channels active
at the receiving node. These numbers should be decimal.
6 7 8 9 10

Decoder matrix for the active channels listed above is:

```
01 00 00 00 00 15 14 12 02 10
00 01 00 00 00 14 14 13 03 11
10 00 01 00 00 12 02 16 02 05
00 00 00 01 00 10 14 05 13 13
00 00 00 00 01 04 13 06 16 06
```

Please enter, on one line and separated by blanks,
the data received on each of the channels active at
the receivers node. The data should be in the form
of octal numbers, and should be entered in order of
increasing channel number.
12 16 07 06 13

the 5 transmitted cleartext words were
(octal numbers expressed in channel order):

12 13 14 15 16

Do you want to decode another 5 words?
(type y or n).

n

Please enter, on one line and separated by a blank,
the field-base and modulus to be used. The
field-base should be a decimal number and the
modulus should be an octal number.

4 23

Please enter the number of channels to be sent
by the transmitting node. This should be a
decimal number.

10

Please enter the number of channels active at
the receivers node; this should be a decimal number.

8

ENF matrix for 8 out of 10
channels over GF $2^{**}(4)$ mod 23 is:

01 00 11 17 16 04 15 11 13 02
 00 01 14 02 07 01 10 04 01 04

please enter, on one line and separated by blanks,
 the numbers of the 8 channels active
 at the receiving node. These numbers should be decimal.
 1 2 3 4 6 7 8 10

Decoder matrix for the active channels listed above is:

01 00 00 00 00 00 00 00 00 00
 00 01 00 00 00 00 00 00 00 00
 00 00 01 00 00 00 00 00 00 00
 00 00 00 01 00 00 00 00 00 00
 14 15 05 01 01 10 01 07 00 12
 00 00 00 00 00 01 00 00 00 00
 00 00 00 00 00 00 01 00 00 00
 00 00 00 00 00 00 00 01 00 00

please enter, on one line and separated by blanks,
 the data received on each of the channels active at
 the receivers node. The data should be in the form
 of octal numbers, and should be entered in order of
 increasing channel number.

10 11 12 13 15 16 17 14

the 8 transmitted cleartext words were
 (octal numbers expressed in channel order):

10 11 12 13 14 15 16 17

do you want to decode another 8 words?

(type y or n).

Appendix F

Copy of Yeh/Reed/Truong paper
on systolic multipliers for finite fields

Systolic Multipliers for Finite Fields $GF(2^m)$

C.-S. YEH, STUDENT MEMBER, IEEE, IRVING S. REED, FELLOW, IEEE, AND T. K. TRUONG, MEMBER, IEEE

Abstract—Two systolic architectures are developed for performing the product-sum computation $AB + C$ in the finite field $GF(2^m)$ of 2^m elements, where A , B , and C are arbitrary elements of $GF(2^m)$. The first multiplier is a serial-in, serial-out one-dimensional systolic array, while the second multiplier is a parallel-in, parallel-out two-dimensional systolic array. The first multiplier requires a smaller number of basic cells than the second multiplier. The second multiplier needs less average time per computation than the first multiplier if a number of computations are performed consecutively. To perform single computations both multipliers require the same computational time. In both cases the architectures are simple and regular and possess the properties of concurrency and modularity. As a consequence they are well suited for use in VLSI systems.

Index Terms—Finite field, logic design, primitive element, systolic array.

I. INTRODUCTION

FINITE or Galois fields have many important and practical applications. Finite fields can be applied to error-correcting codes [1]–[3], switching theory [4], and digital signal processing [5]. For example, finite fields are used in the construction of many error-correcting codes. Reed-Solomon (RS) codes utilize the finite field $GF(2^m)$ of 2^m elements, where m is a positive integer. The encoding and decoding algorithm of a binary RS code require algebraic operations in some field $GF(2^m)$, rather than the usual binary arithmetic operations.

The operations of addition and multiplication in a finite field are quite different from the usual binary arithmetic operations. Because of their simplicity and practical usefulness, only the finite fields $GF(2^n)$ are considered in this paper. Addition in $GF(2^n)$ is bit independent and straightforward. Thus, it is easier than the usual binary addition. On the contrary, multiplication in $GF(2^n)$ is more complex and difficult than binary integer multiplication.

Several circuits have been proposed [1]–[3], [6]–[8] to realize multiplication in $GF(2^n)$. Unfortunately, these circuits are not suited for use in VLSI systems, due to irregular interrouting and complicated control problems as well as a nonmodular structure or lack of concurrency [9].

In this paper two parallel architectures are designed to perform multiplication in $GF(2^n)$. In Section II an algorithm

is derived for multiplication in $GF(2^n)$. This algorithm is mapped into the hardware design in Sections III and IV. In Section III a one-dimensional systolic multiplier for $GF(2^n)$ is designed. This multiplier is serial-in, serial-out. In Section IV, a parallel-in, parallel-out multiplier in $GF(2^n)$ is developed. The latter multiplier has a two-dimensional array structure.

II. MULTIPLICATION IN $GF(2^n)$

It is assumed that the reader is familiar with the basic concepts of finite fields. The properties of finite fields are covered in detail in [1]–[3]. In the following the properties of finite fields are reviewed briefly as required.

A finite field must contain p^m elements, where p is a prime integer and m is a positive integer. The finite field $GF(2^n)$ contains 2^n elements. $GF(2^n)$ is an extension field of the ground field $GF(2)$ of 2 elements, i.e., $GF(2) = \{0, 1\}$. All arithmetic operations in $GF(2^n)$ are performed by taking the results modulo 2.

The nonzero elements of $GF(2^n)$ are generated by a primitive element α , where α is a root of a primitive irreducible polynomial $F(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_1x + f_0$ over $GF(2)$. For example $F(x) = x^4 + x + 1$ is one such primitive irreducible polynomial for $GF(2^4)$.

The nonzero elements of $GF(2^n)$ can be represented as the powers of α , i.e., $GF(2^n) = \{0, \alpha^1, \alpha^2, \dots, \alpha^{2^n-2}, \alpha^{2^n-1} = 1\}$. Since $F(\alpha) = 0$, $\alpha^m = f_{m-1}\alpha^{m-1} + \cdots + f_1\alpha + f_0$. Therefore, an element of $GF(2^n)$ can be also expressed as a polynomial of α with degree less than m . That is, $GF(2^n) = \{a_{m-1}\alpha^{m-1} + \cdots + a_1\alpha + a_0, a_i \in GF(2) \text{ for } 0 \leq i \leq m-1\}$. In the following discussion, the polynomial representation is used to represent the finite field $GF(2^n)$.

Let $A = a_{m-1}\alpha^{m-1} + \cdots + a_1\alpha + a_0$ and $B = b_{m-1}\alpha^{m-1} + \cdots + b_1\alpha + b_0$ be two elements in $GF(2^n)$. Then $A + B = S_{m-1}\alpha^{m-1} + \cdots + S_1\alpha + S_0$, where $S_i = a_i + b_i \pmod{2}$ for $0 \leq i \leq m-1$. Therefore, addition in $GF(2^n)$ is realized easily by m independent EXCLUSIVE-OR gates.

Suppose $P = p_{m-1}\alpha^{m-1} + \cdots + p_1\alpha + p_0$ is the product of A and B , i.e., $P = AB$. P can be written as follows [1]–[3]:

$$P = \sum_{k=0}^{m-1} (A\alpha^k)b_k = \sum_{k=0}^{m-1} \left(\sum_{n=0}^{m-1} a_n\alpha^n \right) b_k \\ = \sum_{n=0}^{m-1} \left(\sum_{k=0}^{m-1} a_n b_k \right) \alpha^n \quad (1)$$

Manuscript received January 10, 1983; revised April 25, 1983. This work was supported in part by the U.S. Air Force Office of Scientific Research under AFOSR-800151 and in part by NASA under Contract NAS7-106.

C.-S. Yeh and I. S. Reed are with the Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089.

T. K. Truong is with the Communication Systems Research Section, Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA 91109.

where $a_n^{(k)}$ is the coefficient of α^n in $A\alpha^k$, i.e., $A\alpha^k = a_{m-1}^{(k)}\alpha^{m-1} + \dots + a_1^{(k)}\alpha + a_0^{(k)}$ for $0 \leq k \leq m-1$. From (1), one obtains $p_n = a_n^{(0)}b_0 + a_n^{(1)}b_1 + \dots + a_n^{(m-2)}b_{m-2} + a_n^{(m-1)}b_{m-1}$.

The computation of $A\alpha^k$ can be performed recursively on k for $0 \leq k \leq m-1$. Initially, for $k=0$, $A\alpha^0 = A$, i.e., $a_n^{(0)} = a_n$ for $0 \leq n \leq m-1$. For $1 \leq k \leq m-1$,

$$A\alpha^k = (A\alpha^{k-1})\alpha = \sum_{n=0}^{m-1} a_n^{(k-1)}\alpha^{n+1} = a_{m-1}^{(k-1)}\alpha^m + \sum_{n=1}^{m-1} a_{n-1}^{(k-1)}\alpha^n. \quad (2)$$

Substituting $\alpha^m = f_{m-1}\alpha^{m-1} + \dots + f_1\alpha + f_0$ into (2) yields

$$A\alpha^k = \sum_{n=1}^{m-1} (a_{n-1}^{(k-1)} + a_{m-1}^{(k-1)}f_n)\alpha^n + a_{m-1}^{(k-1)}f_0. \quad (3)$$

From (3), one obtains

$$\begin{aligned} a_n^{(k)} &= a_{n-1}^{(k-1)} + a_{m-1}^{(k-1)}f_n \quad \text{for } 1 \leq n \leq m-1 \\ a_0^{(k)} &= a_{m-1}^{(k-1)}f_0. \end{aligned} \quad (4)$$

Fig. 1 illustrates the step-by-step operations of a procedure for performing $P = AB + C$ in $GF(2^4)$. In Fig. 1 $a_n^{(k)}$, b_n , c_n , f_n , and p_n are the n th bits of $A\alpha^k$, B , C , F , and P , respectively, where F is the primitive irreducible polynomial. $p_n^{(i)}$ is the partial sum of p_n .

In the following sections this procedure is mapped into two systolic architectures. The above symbols (e.g., F , P , $a_n^{(k)}$) are still used in the following sections.

III. A SERIAL-IN, SERIAL-OUT SYSTOLIC MULTIPLIER FOR $GF(2^m)$

In this section a one-dimensional systolic array is developed to compute $P = AB + C$ in $GF(2^m)$. A similar structure was proposed to multiply the usual two's complement binary numbers [10]. For simplicity in description the ensuing discussion is limited to the particular finite field $GF(2^4)$. In Fig. 2 this architecture is shown for $GF(2^4)$. The primitive irreducible polynomial is $F = f_3\alpha^3 + f_2\alpha^2 + f_1\alpha + f_0$. Input d_n receives the b_n of B . The n th bits c_n , a_n , and f_n of C , A , and F are received serially at inputs e_0 , g_0 , and h_0 , respectively. Two control signals, START and END, are used in the design. Inputs r_0 and t_0 receive START and END control signals, respectively.

Output e_n serially transmits the n th bit p_n of the result P out of the system. The order of the inputs and outputs are also shown in Fig. 2. The flip-flops associated with inputs t_0 and h_0 are used for the purpose of synchronization.

The circuit of cell L_i is shown in Fig. 3. The operation of flip-flops in this system is synchronized implicitly by a clock signal. In Fig. 3, when $r_i^* = 1$, $u_i = g_i^*$ at the next time unit (through switch SW). When $r_i^* = 0$, u_i sustains its value. Two principle operations of the system are the following:

$$\begin{aligned} e_{i+1} &\leftarrow (g_i^*d_i) \oplus e_i^* \\ g_{i+1}^* &\leftarrow (u_ih_i^*) \oplus (g_i^*t_i^*) \end{aligned} \quad (5)$$

STEP NUMBER	OPERATIONS
1	$p_3^{(0)} = c_3$ $a_3^{(0)} = a_3$
2	$p_3^{(1)} = p_3^{(0)} + a_3^{(0)}b_0$ $a_2^{(1)} = a_2$ $p_2^{(0)} = c_2$
3	$p_3^{(1)} = p_3^{(0)} + a_3^{(0)}b_0$ $a_1^{(1)} = a_1$ $p_2^{(1)} = p_2^{(0)} + a_2^{(0)}b_0$ $a_3^{(1)} = a_3^{(0)} + a_3^{(0)}b_1$ $p_1^{(0)} = c_1$
4	$p_3^{(2)} = p_3^{(1)} + a_3^{(1)}b_1$ $a_0^{(1)} = a_0$ $p_2^{(1)} = p_2^{(0)} + a_2^{(0)}b_0$ $a_2^{(1)} = a_2^{(0)} + a_2^{(0)}b_1$ $p_1^{(1)} = p_1^{(0)} + a_1^{(0)}b_0$
5	$p_3^{(2)} = p_3^{(1)} + a_3^{(1)}b_1$ $a_1^{(1)} = a_1^{(0)} + a_1^{(0)}b_1$ $p_2^{(2)} = p_2^{(1)} + a_2^{(1)}b_1$ $a_0^{(1)} = a_0^{(0)} + a_0^{(0)}b_1$ $p_1^{(2)} = p_1^{(1)} + a_1^{(1)}b_1$
6	$p_3^{(3)} = p_3^{(2)} + a_3^{(2)}b_2$ $a_0^{(2)} = a_0^{(1)} + a_0^{(1)}b_2$ $p_2^{(2)} = p_2^{(1)} + a_2^{(1)}b_1$ $a_1^{(2)} = a_1^{(1)} + a_1^{(1)}b_2$ $p_1^{(2)} = p_1^{(1)} + a_1^{(1)}b_1$
7	$p_3^{(3)} = p_3^{(2)} + a_3^{(2)}b_2$ $a_0^{(2)} = a_0^{(1)} + a_0^{(1)}b_2$ $p_2^{(3)} = p_2^{(2)} + a_2^{(2)}b_2$ $a_1^{(2)} = a_1^{(1)} + a_1^{(1)}b_2$ $p_1^{(3)} = p_1^{(2)} + a_1^{(2)}b_2$
8	$p_3^{(4)} = p_3^{(3)} + a_3^{(3)}b_3$ $a_0^{(3)} = a_0^{(2)} + a_0^{(2)}b_3$ $p_2^{(3)} = p_2^{(2)} + a_2^{(2)}b_1$ $a_1^{(3)} = a_1^{(2)} + a_1^{(2)}b_3$ $p_1^{(3)} = p_1^{(2)} + a_1^{(2)}b_1$
9	$p_3^{(4)} = p_3^{(3)} + a_3^{(3)}b_3$ $a_0^{(3)} = a_0^{(2)} + a_0^{(2)}b_3$ $p_2^{(4)} = p_2^{(3)} + a_2^{(3)}b_3$ $a_1^{(3)} = a_1^{(2)} + a_1^{(2)}b_3$ $p_1^{(4)} = p_1^{(3)} + a_1^{(3)}b_3$
10	$p_3^{(4)} = p_3^{(3)} + a_3^{(3)}b_3$ $a_0^{(3)} = a_0^{(2)} + a_0^{(2)}b_3$ $p_2^{(4)} = p_2^{(3)} + a_2^{(3)}b_3$ $a_1^{(3)} = a_1^{(2)} + a_1^{(2)}b_3$ $p_1^{(4)} = p_1^{(3)} + a_1^{(3)}b_3$
11	$p_0^{(4)} = p_0^{(3)} + a_0^{(3)}b_3$

Fig. 1. A procedure for computing $P = AB + C$ in the finite field $GF(2^4)$ where A , B , C , and P are elements of $GF(2^4)$.

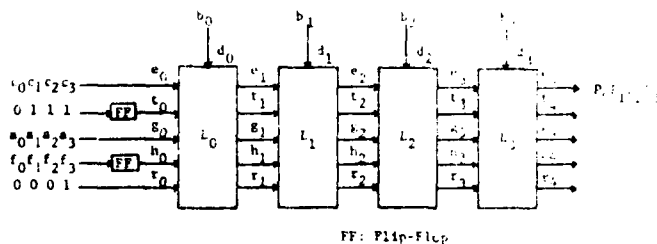


Fig. 2. A serial-in, serial-out systolic multiplier for the finite field $GF(2^4)$.

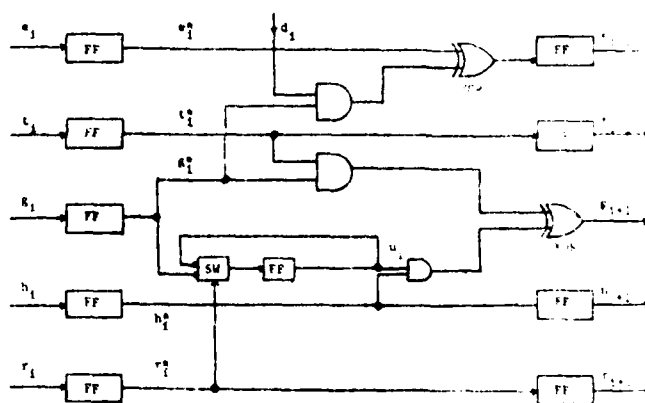


Fig. 3. The circuit of the cell L_i used in the multiplier shown in Fig. 2.

TABLE I
SOME PROPERTIES OF TWO SYSTOLIC MULTIPLIERS FOR $GF(2^m)$

Property	Multipplier	The multiplier in Fig. 2	The multiplier in Fig. 4
1. Number of basic cells		m	m^2
2. Input/output format		serial	parallel
3. Minimum average time per computation (time units)		m	1
4. Delay between first input and first output of a computation (time units)		$2m$	$2m$ ($3m$ if input/output delay is also counted)
5. Number of control signals		2	0

where $0 \leq i \leq 3$, \oplus denotes EXCLUSIVE-OR operation, and the backwards arrow denotes the substitution operation.

A comparison of the procedure in Fig. 1 and the structure in Figs. 2 and 3 yields the following facts. The signal u_i in L_i is equal to $a_i^{(i)}$ in $A\alpha^i$. The signal g_i^* is equal to $a_n^{(i)}$ in $A\alpha^i$ for some n . The signal e_i^* is equal to the partial sum of $AB + C$.

The multiplier in Fig. 2 can be generalized to the finite field $GF(2^m)$ by simply concatenating m identical cells. Extra registers and control signals are required if the b_i 's are inputted serially into the system in the same order as the a_i 's. Some properties of this multiplier are listed in Table I.

IV. A PARALLEL-IN, PARALLEL-OUT MULTIPLIER FOR $GF(2^m)$

In this section a parallel-in, parallel-out, two-dimensional systolic array is designed for performing $P = AB + C$ in $GF(2^m)$. A similar structure was designed [11] to perform multiplications in standard binary arithmetic. The discussion in this section is again limited to the finite field $GF(2^4)$. An analogous development can be constructed for any other finite field $GF(2^m)$. Fig. 4 shows this multiplier for $GF(2^4)$. In Fig. 4 D^n denotes an n -bit shift register or delay device. Inputs $d_{n,0}$'s, $e_{n,0}$'s, $g_{n,0}$'s, and $h_{n,0}$'s receive in parallel the b_n 's of B , c_n 's of C , a_n 's of A , and f_n 's of F , respectively, for $0 \leq n \leq 3$. The p_n 's of the result P are transmitted out the system in parallel from outputs $e_{n,4}$'s for $0 \leq n \leq 3$.

The circuit of a basic cell L_{ij} is shown in Fig. 5. This circuit is similar to the circuit shown in Fig. 3. Two of the primary operations of a basic cell are the same as the operations given in (5). One may use degenerative versions of the circuit in Fig. 5 for the cells in the bottom row and the rightmost column of the array structure in Fig. 4 since some inputs and outputs of these cells are not used. Note that the signal $g_{n,4}$ is equal to the $a_n^{(4)}$ of $A\alpha^4$. The signal $u_{n,4}$ is equal to $a_n^{(4)}$ of $A\alpha^4$ for $0 \leq n \leq 3$.

Some properties of the multiplier in Fig. 4 are also listed in Table I. The multiplier in Fig. 4 is "programmable" since F is changeable. If F is fixed or seldom changed then the design can be simplified by eliminating all flip-flops associated with $h_{n,0}$. For such a case buffers and long wires may be required.

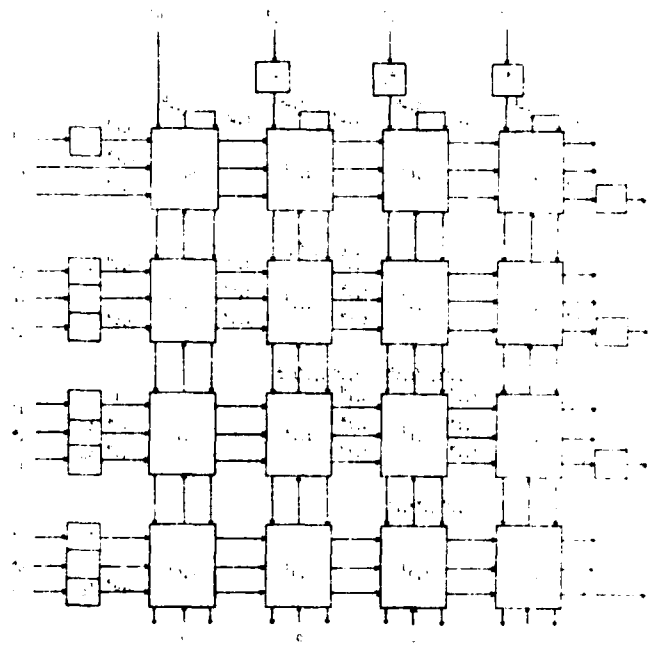


Fig. 4. A two-dimensional parallel-in, parallel-out systolic multiplier for the finite field $GF(2^4)$.

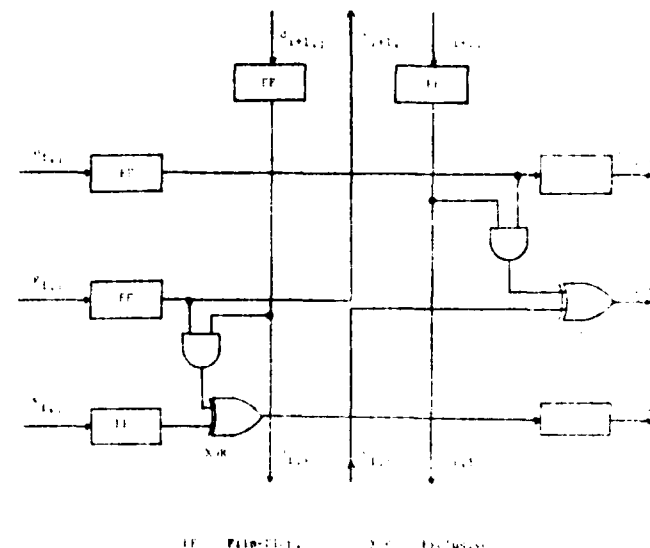


Fig. 5. The circuit of basic cell L_{ij} used in the multiplier shown in Fig. 4.

V. CONCLUSION

Two parallel architectures are designed for performing multiplication in the finite field $GF(2^m)$ of 2^m elements. A comparison between these two multipliers is listed in Table I. The multiplier in Fig. 4 can be viewed as a "time expansion" of the multiplier in Fig. 2. Both multipliers are suited well for VLSI systems because of the simple control, the regular interconnection pattern, the modular structure, and finally the complete concurrency of their operations.

ACKNOWLEDGMENT

The authors would like to thank the referees for several useful suggestions.

REFERENCES

- [1] P. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. Amsterdam: North-Holland, 1978.
- [2] W. W. Peterson and E. J. Weldon, Jr., *Error-Correcting Codes*, 2nd ed. Cambridge, MA: MIT Press, 1972.
- [3] E. R. Berlekamp, *Algebraic Coding Theory*. New York: McGraw-Hill, 1968.
- [4] B. Benjathrit and I. S. Reed, "Galois switching functions and their applications," *IEEE Trans. Comput.*, vol. C-25, pp. 78-86, Jan. 1976.
- [5] I. S. Reed and T. K. Truong, "The use of finite fields to compute convolutions," *IEEE Trans. Inform. Theory*, vol. IT-21, no. 2, pp. 208-213, Mar. 1975.
- [6] T. C. Bartee and D. I. Schneider, "Computation with finite fields," *Inform. Contr.*, vol. 6, pp. 79-98, Mar. 1963.
- [7] B. A. Laws and C. K. Rushforth, "A cellular-array multiplier for GF(2^m)," *IEEE Trans. Comput.*, vol. C-20, pp. 1573-1578, Dec. 1971.
- [8] R. G. Gallager, *Information Theory and Reliable Communication*. New York: Wiley, 1968.
- [9] H. T. Kung, "Why systolic architectures?" *IEEE Computer*, vol. 15, pp. 37-46, Jan. 1982.
- [10] R. E. Lyon, "Two's complement pipeline multipliers," *IEEE Trans. Commun.*, vol. COM-24, pp. 418-425, Apr. 1976.
- [11] J. V. McCanny and J. G. McWhirter, "Completely iterative pipelined multiplier array suitable for VLSI," *IEE Proc. G, Electronic Circuits and Systems*, vol. 129, no. 2, pp. 40-46, Apr. 1982.



C.-S. Yeh (S'79) was born in Tainan, Taiwan, on May 13, 1951. He received the B.S. degree in electrical engineering from National Cheng Kung University, Taiwan, in 1974, the M.S. degree in electronics from National Chiao Tung University in 1976, and the M.S.E.E. degree in computer engineering in 1981 from the University of Southern California (USC), Los Angeles, where he is now a Ph.D. candidate.

Since 1979 he has been a Teaching Research

Assistant at USC. His research interests are computer systems, VLSI architecture, and digital signal processing.



Irving S. Reed (SM'69-F'73) was born in Seattle, WA on November 12, 1923. He received the B.S. and Ph.D. degrees in mathematics from the California Institute of Technology, Pasadena, in 1944 and 1949, respectively.

From 1951 to 1960 he was associated with Lincoln Laboratory, Massachusetts Institute of Technology, Lexington. From 1960 to 1968 he was a Senior Staff Member with the RAND Corporation, Santa Monica, CA. Since 1963 he has been a Professor of Electrical Engineering and Computer Science at the University of Southern California, Los Angeles. He holds the Charles Lee Powell Professorship in Computer Engineering at USC. He is also a Consultant to the RAND Corporation, the MITRE Corporation, and a Director of Adaptive Sensors, Inc. His interests include mathematics, VLSI computer design, coding theory, stochastic processes, and information theory.

Dr. Reed is a member of the National Academy of Engineering.



T. K. Truong (M'82) was born in Cholon, Vietnam, on December 4, 1944. He received the B.S. degree in electrical engineering from the National Cheng Kung University, Taiwan, China, in 1967, the M.S. degree in electrical engineering from Washington University, St. Louis, MO, in 1971, and the Ph.D. degree from the University of Southern California, Los Angeles, in 1976.

Since 1976, he has been with the Communication Systems Research Section, System Engineering Technical Staff of the Jet Propulsion Laboratory, Pasadena, CA. Also, he is currently a part-time Research Scientist at the University of Southern California, and a Consultant to the Department of Radiology, Memorial Hospital of Long Beach, CA. His research interests are in the areas of mathematics, VLSI architecture, coding theory, X-ray reconstruction, and digital signal processing.

Appendix G

Copy of Bloom paper on threshold schemes

A Note on Superfast Threshold Schemes

John R. Bloom

Abstract: Threshold schemes, or key safeguarding schemes, are innovative new approaches to cryptokey transfer or secure data storage problems. This note outlines a class of schemes which approach optimality of speed and simplicity. The schemes are based on linear maps over finite fields. These schemes are the proper generalization of Vernam pads.

Key words and phrases: privacy, security, cryptography, message passing

CR Categories: 3.81, 5.6, 5.25

A threshold scheme is a method for producing, from a message x , n "shadows" y_1, \dots, y_n , with the properties that:

1. Any r shadows suffice to determine x .
2. No $r-1$ shadows give any information about x .

Threshold schemes have been discussed in papers by Blakley [2], Shamir [4], where many applications are discussed, and Asmuth and Bloom [1], where a class of schemes including Shamir's is discussed, and further cross checking capabilities are also introduced. This paper introduces a class of schemes of optimum speed and simplicity when the message length is large compared to r . These schemes are the generalization of Vernam pads.

To generate such a threshold scheme, pick v_0, v_1, \dots, v_n vectors in \mathbb{F}_q^r so that no r are linearly dependent. This can be done if $q > n$, and conjecturally for no smaller q . (See [3] pp. 323-328).

Considering the message x and the shadows y_i to be elements of \mathbb{F}_q , construct a linear map L from \mathbb{F}_q^r to \mathbb{F}_q with $Lv_0 = x$

and Lv_1, \dots, Lv_{r-1} random. Letting $y_i = Lv_i$, $i = 1, \dots, r$ one has produced a threshold scheme. Property 1 is satisfied since any r v_i 's span \mathbb{F}_q^r , and property 2 is satisfied since v_0 is not dependent on any $r-1$ v_i 's.

In practice one picks q as small as possible and reduces x to a sequence of m messages of size q .

Proposition: To produce a sequence of m shadows for fixed i requires at most mr additions and mr multiplications. To reconstitute the sequence of m x 's requires at most $\frac{r^3}{3} + (m+1)r$ additions and $\frac{r^3}{3} + (m+1)r$ multiplications. The algorithm meeting these requirements is described below.

For fixed i , there is a vector $w \in \mathbb{F}_q^r$ with $v_i = \sum_{j=0}^{r-1} w_j v_j$. One can construct $y_i = Lv_i$ from the relation $Lv_i = \sum_{j=0}^{r-1} w_j Lv_j$. To reconstitute x from y_{i1}, \dots, y_{ir} , one solves $\sum_{j=1}^r u_j v_{ij} = v_0$ for the vector u by Gaussian elimination and forms $x = Lv_0$ from the relation $Lv_0 = \sum_{j=1}^r u_j y_{ij}$. This algorithm clearly satisfies the op counts given above.

Since q can be chosen extremely small in many applications, two savings are possible. If the u 's are stored, no Gaussian elimination is necessary. If a table of Zech's logarithms is stored ([3], p. 91) the encoding and decoding algorithms reduce to r additions and r table look-ups.

For large m , these threshold schemes take $(2+o(1))r$ operations. Conjecturally, one cannot have threshold schemes requiring $(2-o(1))r$

operations for large r, n . An elementary result is the following.

Proposition: A threshold scheme cannot be decoded in fewer than r operations.

Proof. Since a threshold scheme requires that no $r-1$ y_i 's determine x , all r shadows must be used.

The definition of a threshold scheme requires that each shadow carry as much information as the message x . This message expansion can be overcome by using a pseudo-threshold scheme. All existing schemes have such variants, only the variant of this paper's superfast scheme is outlined.

Pick $v_{-k}, \dots, v_0, \dots, v_n$ in \mathbb{F}_q^r so that no r are linearly dependent. Form a linear map L with $Lv_{-j} = x_j$ for $j = 0, \dots, K$ where $K \geq r$ and x_0, \dots, x_K are messages or parts of a message x . Let $y_i = Lv_i$ $i = 1, \dots, n$. All other details are as before.

For these schemes one has, for each i that no $r-1$ y_i 's give any information about x_i , and this may suffice for many applications, but $r-1$ y_i 's do give information about the tuple (x_0, \dots, x_K) . In essence, given $r-s$ y_i 's, if one correctly guesses s of the x_i 's the rest follow.

References

- [1] Asmuth, C. and Bloom, J. "A Modular Approach to Key Safeguarding"
- [2] Blakley, G. R. "Safeguarding Cryptographic Keys", Proceedings of the National Computer Conference, 1979, AFIPS Conference Proceedings, vol. 48 (1979) pp. 313-317.
- [3] MacWilliams, F. J., and Sloane, N.J.A. The Theory of Error-Correcting Codes, North-Holland, 1977.
- [4] Shamir, A. "How to share a secret", Communications of the ACM, vol. 22 no. 11, (Nov. 1979) pp. 612-613.

Appendix H

Program for encoding procedure
(including Stages 1, 2 and 4)

```

10 10 Line# Source Line IBM Personal Computer Pascal Compiler V1.00
20 1 program enf (input,output);
20 2
10 3 const WORDLENGTH = 16;
10 4 MAXINDEX = 32;
20 5
10 6 type mat_row = array[1..MAXINDEX] of integer;
10 7 matrix = array[1..MAXINDEX] of mat_row;
10 8 channel_array = array[1..MAXINDEX] of integer;
20 9
10 10
11 11 { the TWO_TO_THE function makes up for the lack of a generalize
11 12 d
12 13 exponentiation operator in standard Pascal. It returns two
13 14 raised to the power of its caller-supplied argument.
13 15 }
20 16
20 17 function TWO_TO_THE (argument : integer): integer;
20 18
20 19 var accumulator : integer;
20 20 index : integer;
20 21
20 22 begin
21 23 accumulator := 1;
21 24 for index := 1 to argument do
21 25 accumulator := accumulator * 2;
21 26 TWO_TO_THE := accumulator;
10 27 end;
20 28
svmtab 29 Offset Length Variable - TWO_TO_THE
- 2 14 Return offset, Frame length
- 6 2 (function return) : Integer
- 0 2 ARGUMENT :Integer ValueP
- 10 2 INDEX :Integer
- 8 2 ACCUMULATOR :Integer
20 30
20 31
20 32 { the READ_OCTAL routine; this routine allows the user of
20 33 the program to input his values in octal rather than in
20 34 decimal; it replaces the Pascal standard "read" routine. }
20 35
20 36 procedure READ_OCTAL (var total : integer);
20 37
20 38 const BLANK = ' ';
20 39
20 40 var inchar : char;

```

READ_OCTAL

J6 IC Line# Source Line IBM Personal Computer Pascal Compiler V1.00

```

37
38
20 39 begin
21 40 read (inchar);
21 41 total := 0;
42
21 43 while (inchar = BLANK) do
21 44 read (inchar);
45
21 46 while not (inchar = BLANK) do begin
22 47 total := total * 8 + (ord(inchar) - ord('0'));
22 48 read (inchar)
22 49 end
50
10 51 end;
```

```

37 51 Offset Length Variable - READ_OCTAL
- 2 8 Return offset, Frame length
- 0 2 TOTAL :Integer Var P
- 6 1 INCHAR :Char
```

```

52
53
54 (the WRITE_OCTAL routine; it replaces the Pascal standard
55 "write" routine and allows the program to report its
56 output values in octal rather than in decimal. )
57
20 58 procedure WRITE_OCTAL (number : integer;
20 59 field_base : integer);
60
20 61 var outbuf : array [1..WORDLENGTH] of char;
20 62 temp : integer;
20 63 index : integer;
64
65
20 66 begin
21 67 for index := 1 to WORDLENGTH do outbuf[index] := '0';
21 68 index := 1;
69
21 70 while (number > 0) do begin
22 71 temp := number mod 8;
22 72 outbuf[index] := chr (ord('0') + temp);
22 73 index := index + 1;
22 74 number := number div 8
21 75 end;
```

WRITE_OCTAL

```

IBM Personal Computer Pascal Compiler V1.00
Line# Source Line
21 77 temp := (field_base + 2) div 3);
21 78 if (temp < 1) then temp := 1;
21 79 if (temp < (index - 1)) then temp := index - 1;
21 80 for index := temp downto 1 do write(outbuf[index]);
21 81 write(' ');
21 82
10 83 end;

Svmtab 83 Offset Length Variable - WRITE_OCTAL
- 4 30 Return offset, Frame length
- 0 2 NUMBER :Integer ValueP
- 24 2 TEMP :Integer
- 26 2 INDEX :Integer
- 2 2 FIELD_BASE :Integer ValueP
- 22 16 OUTBUF :Array

94
95
96
97
98 ( The ADD function returns the logical xor of its two caller-
99 supplied arguments. This is addition over GF(n) for any n. )
100
20 90 function ADD (term1 : integer;
20 91 term2 : integer) : integer;
20 92
20 93 begin
= 21 94 ADD := ( (term1 or term2) and (not(term1 and term2)) )
10 95 end;

Svmtab 95 Offset Length Variable - ADD
- 4 10 Return offset, Frame length
- 8 2 (function return) : Integer
- 0 2 TERM1 :Integer ValueP
- 2 2 TERM2 :Integer ValueP

96
97
98 ( The MULTIPLY function performs multiplication over GF(n)
99 modulo the caller-supplied modulus and returns the result
100 of the multiplication. )
101
20 102 function MULTIPLY (factor1 : integer;
20 103 factor2 : integer;
20 104 modulus : integer;
20 105 field_base : integer) : integer;
106
MULTIPLY

```

```

05 10 Line# Source Line IBM Personal Computer Pascal Compiler V1.00
20 107 var index : integer;
20 108 answer : integer;
20 109
20 110 begin
20 111
21 112 answer := 0;
20 113
21 114 for index := 0 to (field_base - 1) do
21 115 begin
22 116 answer := answer * 2;
20 117
22 118 if (( (factor1 mod TWO_TO_THE (field_base - index))
22 119 div TWO_TO_THE (field_base - (index+1)) ) > 0)
22 120 then answer := ADD (answer, factor2);
20 121
22 122 if ( (answer div TWO_TO_THE (field_base)) > 0)
22 123 then answer := ADD (answer,
22 124 TWO_TO_THE (field_base) + modulus)
21 125 end;
= 21 126 MULTIPLY := answer
10 127 end;

Symtab 127 Offset Length Variable - MULTIPLY
- 8 20 Return offset, Frame length
- 12 2 (function return) : Integer
- 0 2 FACTOR1 :Integer ValueP
- 14 2 INDEX :Integer
- 16 2 ANSWER :Integer
- 2 2 FACTOR2 :Integer ValueP
- 4 2 MODULUS :Integer ValueP
- 6 2 FIELD_BASE :Integer ValueP

128
129
130 ( The INVERSE function. It accepts a field element and
131 returns the element's multiplicative inverse. This
132 implementation is very slow & primitive-- it should be
133 replaced by Davida's inverse routine or some other fast
134 implementation at the first opportunity. )
135
20 136 function INVERSE ( element : integer;
20 137 field_base : integer;
20 138 modulus : integer ): integer;
20 139
20 140 var index : integer;
20 141 answer : integer;

```

INVERSE

```

IBM Personal Computer Pascal Compiler V1.00
Line# Source Line
20 142 squares : integer;
    143
20 144 begin
    145
21 146     answer := 1;
21 147     squares := element;
    148
21 149     for index := 1 to (field_base - 1) do
21 150         begin
22 151             squares := MULTIPLY (squares,squares,modulus,field_base);
22 152             answer  := MULTIPLY (answer,squares,modulus,field_base)
21 153         end;
    154
- 21 155     INVERSE := answer
    156
10 157 end;

```

```

Syntab 157 Offset Length Variable - INVERSE
-      6      20 Return offset, Frame length
-     10       2 (function return) : Integer
-      0       2 ELEMENT :Integer ValueP
-     12       2 INDEX :Integer
-     14       2 ANSWER :Integer
-      2       2 FIELD_BASE :Integer ValueP
-      4       2 MODULUS :Integer ValueP
-     16       2 SQUARES :Integer

```

158

159 (The DIVIDE function performs Galois-field division.

160 it accepts dividend, divisor, modulus, and field-base

161 (in that order), takes the inverse of the divisor, and

162 multiplies the result by the dividend.

163

```

20 164 function DIVIDE ( dividend : integer;
20 165                  divisor   : integer;
20 166                  modulus    : integer;
21 167                  field_base : integer ): integer;
    168
20 169 var divisor_inverse : integer;
    170
20 171 begin
    172
21 173     divisor_inverse := INVERSE (divisor, field_base, modulus);
- 21 174     DIVIDE := MULTIPLY (dividend, divisor_inverse,
21 175                        modulus, field_base
    176

```

DIVIDE

15 10 Line# Source Line
16 177 end;

Source Line	Offset	Length	Variable - DIVIDE	
177	- 8	16	Return offset, Frame length	
	- 12	2	(function return) :	Integer
	- 0	2	DIVIDEND	:Integer ValueP
	- 2	2	DIVISOR	:Integer ValueP
	- 4	2	MODULUS	:Integer ValueP
	- 6	2	FIELD_BASE	:Integer ValueP
	- 14	2	DIVISOR_INVERSE	:Integer

178

179

180 { The CONSTRUCT_VAN routine. This procedure constructs a
181 square vandermonde matrix with the dimension supplied by the
182 calling routine. }
183

```
20 184 procedure CONSTRUCT_VAN (var van      : matrix;
20 185                          n            : integer;
20 186                          field_base  : integer;
20 187                          modulus    : integer );
20 188
```

```
20 189   var   row      : integer;
20 190        column   : integer;
20 191        exponent : integer;
20 192        index    : integer;
20 193        temp     : integer;
20 194
```

```
20 195   begin
```

```
21 196       if (n < 3) then writeln ('van dimension < 3: error')
21 197       else
21 198         begin
```

```
200           (build first row of van. )
```

```
201           van [1][1] := 1;
202           for column := 2 to n do
203             van [1][column] := 0;
```

```
204           (build second row of van. )
```

```
205           for column := 1 to n do
206             van [2][column] := 1;
```

```
207           (build third row of van. )
```

CONSTRUCT_VAN

```

06 10  Line#   Source Line      IBM Personal Computer Pascal Compiler V1.00
      213
22    214       van [3][1] := 1;
22    215       van [3][2] := 2;
22    216       for column := 3 to n do
22    217         van [3][column] :=
22    218           MULTIPLY (van [3][column - 1], 2,
22    219             modulus, field_base
      220           );
      221       (build remaining rows of van.)
      222
22    223       if (n > 3) then
22    224         for row := 4 to n do begin
22    225           van [row][1] := 1;
22    226           for column := 2 to n do
22    227             van [row][column] :=
22    228               MULTIPLY (van [row - 1][column], van [3][column
22    228             ],
22    229               modulus, field_base
22    229             );
22    230         end
      231
      232
22    233       end
10    234     end;

```

Symbol	Offset	Length	Variable - CONSTRUCT_VAN	
	- 8	32	Return offset, Frame length	
	- 0	2	VAN	:Array VarP
	- 2	2	N	:Integer ValueP
	- 12	2	ROW	:Integer
	- 4	2	FIELD_BASE	:Integer ValueP
	- 6	2	MODULUS	:Integer ValueP
	- 14	2	COLUMN	:Integer
	- 18	2	INDEX	:Integer
	- 20	2	TEMP	:Integer
	- 16	2	EXPONENT	:Integer

235

236

```

237  ( the BUILD_ENF routine.  It accepts the modulus and field-base
238  desired by the user and the number of channels to be
239  transmitted and produces a CODING-NORMAL-FORM matrix (enf).
240  This matrix is (transmitted) X (transmitted - 2), and is
241  gotten by column-reducing the first (transmitted - 2) columns
242  of a (transmitted) X (transmitted) Vandermonde matrix so
243  that the resulting matrix is upper-right triangular.

```

CONSTRUCT_VAN

```

08 IC Line# Source Line IBM Personal Computer Pascal Compiler V1.00
    243 }
    244
20   245 procedure BUILD_ENF (var enf          : matrix;
20   246                      transmitted : integer;
20   247                      modulus    : integer;
20   248                      field_base  : integer );
    249
20   250 var   columns      : integer;
20   251         rows       : integer;
20   252         reducing_col : integer;
20   253         reduced_elt : integer;
20   254         row         : integer;
20   255         column      : integer;
20   256         dimension   : integer;
    257
    258
20   259 begin
    260
21   261     dimension := TWO_TO_THE (field_base);
    262
21   263     CONSTRUCT_VAN (enf, dimension, field_base, modulus);
    264
21   265     rows := dimension;
21   266     columns := dimension - 2;
21   267     for reducing_col := 1 to columns do begin
    268
21   269       ( divide reducing_col through by its lead element (we want
21   270         ones along the diagonal.)
    271
22   272         for row := 1 to (rows - reducing_col) do
22   273           enf[row][reducing_col] :=
22   274             DIVIDE (enf[row][reducing_col],
22   275                     enf[(rows-reducing_col)+1][reducing_col],
22   276                     modulus, field_base);
22   277           enf[(rows-reducing_col)+1][reducing_col] := 1;
    278
22   279       ( column-reduce to clear the row containing the lead element of
22   280         reducing_col (that lead element is now a 1).
22   281
22   282         if (reducing_col < columns) then
22   283           for column := (reducing_col + 1) to columns do begin
22   284             reduced_elt := enf[(rows-reducing_col)+1][column];
22   285             for row := 1 to (rows - reducing_col) do
22   286               enf[row][column] :=
22   287                 ADD (enf[row][column],
BUILD_ENF

```

Page 9
05-24-84
00:17:41

```

10 10 Line# Source Line      IBM Personal Computer Pascal Compiler V1.00
23 268      MULTIPLY ( reduced_elt, enf[row][reducing_c
23 268      o1],
23 269      modulus, field_base ,
23 269      );
23 290      enf[(rows-reducing_col)+1][column] := 0
23 291      end
23 292
22 293      end
10 294      end;

```

Symtab	294	Offset	Length	Variable - BUILD_ENF	
		- 8	36	Return offset, Frame length	
		- 0	2	ENF	:Array VarP
		- 2	2	TRANSMITTED	:Integer ValueP
		- 4	2	MODULUS	:Integer ValueP
		- 12	2	COLUMNS	:Integer
		- 14	2	ROWS	:Integer
		- 20	2	ROW	:Integer
		- 22	2	COLUMN	:Integer
		- 6	2	FIELD_BASE	:Integer ValueP
		- 24	2	DIMENSION	:Integer
		- 18	2	REDUCED_ELT	:Integer
		- 16	2	REDUCING_COL	:Integer

```

295
296
297 ( the TRANSPOSE routine accepts a matrix and its dimensions
298 and produces the transpose of the matrix. )
299
20 300 procedure TRANSPOSE (      m      : matrix;
20 301                        var m_prime : matrix;
20 302                        m_rows  : integer;
20 303                        m_cols  : integer );
20 304
20 305 var row : integer;
20 306     col : integer;
20 307
20 308 begin
21 309     for row := 1 to m_rows do
21 310         for col := 1 to m_cols do
21 311             m_prime[col][row] := m[row][col];
10 312     end;

```

Symtab	312	Offset	Length	Variable - TRANSPOSE	
		- 2054	2066	Return offset, Frame length	
		- 2046	2048	M	:Array ValueP

TRANSPOSE

IS	IC	Line#	Source Line	IBM Personal Computer Pascal Compiler V1.00
			- 2048 2 M_PRIME	:Array VarP
			- 2050 2 M_ROWS	:Integer ValueP
			- 2052 2 M_COLS	:Integer ValueP
			- 2058 2 ROW	:Integer
			- 2060 2 COL	:Integer

313

314

315 (the EXTRACT_SUBMATRIX routine accepts the enf matrix, the
 316 number of channels to be transmitted, and the number of
 317 channels to be received. It produces a smaller matrix
 318 which will be used to construct the encode and decode keys
 319 for this particular configuration of transmitted and
 320 received channels.)

321

```

20 322 procedure EXTRACT_SUBMATRIX (var submatrix      : matrix;
20 323                             enf                  : matrix;
20 324                             transmitted          : integer;
20 325                             received            : integer;
20 326                             field_base         : integer );
20 327

```

```

20 328 var  enf_prime      : matrix;
20 329      row            : integer;
20 330      column         : integer;
20 331      dimension      : integer;
20 332      index          : integer;
20 333

```

334

```

20 335 begin
20 336
21 337     dimension := TWO_TO_THE (field_base);
21 338
21 339     TRANSPOSE (enf, enf_prime, dimension, dimension-2);
21 340
21 341     index := 0;
21 342
21 343     for row := (dimension - (transmitted - 1)) to
21 344         (dimension - received) do begin
22 345         index := index + 1;
22 346         for column := 1 to transmitted do
22 347             submatrix[index][column] := enf_prime[row][column]
22 348         end
22 349
10 350 end;

```

Symtab 350 Offset Length Variable - EXTRACT_SUBMATRIX

EXTRACT_SUBMATRIX

Line#	Source Line	IBM Personal Computer Pascal Compiler V1.00		
- 2056	4122	Return offset, Frame length		
- 0	2	SUBMATRIX	:Array	VarP
- 2048	2048	ENF	:Array	ValueP
- 2052	2	RECEIVED	:Integer	ValueP
- 4108	2	ROW	:Integer	
- 4110	2	COLUMN	:Integer	
- 4114	2	INDEX	:Integer	
- 4106	2048	ENF_PRIME	:Array	
- 4112	2	DIMENSION	:Integer	
- 2050	2	TRANSMITTED	:Integer	ValueP
- 2054	2	FIELD_BASE	:Integer	ValueP

351

352

353 (the BUILD_ENCODE_KEY builds the matrix which will be used
 354 to produce the (transmitted - received) coded channels for
 355 transmission. The first (received) channels are sent in
 356 the clear.)
 357

```

20 358 procedure BUILD_ENCODE_KEY (var encode_key      : matrix;
20 359                             submatrix          : matrix;
20 360                             transmitted          : integer;
20 361                             received            : integer;
20 362                             modulus             : integer;
20 363                             field_base         : integer );
20 364
20 365 var columns      : integer;
20 366     rows         : integer;
20 367     col          : integer;
20 368     row          : integer;
20 369     reducing_row  : integer;
20 370     reduced_elt   : integer;
20 371
20 372
20 373 begin
21 374     rows := transmitted - received;
21 375     columns := transmitted;
21 376
21 377     for reducing_row := rows downto 2 do
21 378         for row := (reducing_row - 1) downto 1 do begin
21 379
22 380             reduced_elt := submatrix[row]
22 381                             [received+(rows-reducing_row)+1];
22 382             for col := 1 to (received+(rows-reducing_row)) do
22 383                 submatrix [row][col] :=
22 384                     ADD (submatrix[row][col],

```

BUILD_ENCODE_KEY

```

00 10  Line#  Source Line      IBM Personal Computer Pascal Compiler V1.00
22   385      MULTIPLY (reduced_elt, submatrix[reducing_row][
22   385      col],
22   386      modulus, field_base)
22   386      );
22   387      submatrix[row][received+(rows-reducing_row)+1] := 0
22   388
21   389      end;
22   390
21   391      for row := 1 to rows do
21   392      for col := 1 to columns do
21   393      encode_key[row][col] := submatrix[row][col]
22   394
10   395      end;

```

```

395  Offset Length  Variable - BUILD_ENCODE_KEY
- 2058  2084  Return offset, Frame length
-      0      2  ENCODE_KEY           :Array   VarP
- 2048  2048  SUBMATRIX           :Array   ValueP
- 2052      2  RECEIVED           :Integer ValueP
- 2054      2  MODULUS           :Integer ValueP
- 2062      2  COLUMNS           :Integer
- 2064      2  ROWS           :Integer
- 2066      2  COL           :Integer
- 2068      2  ROW           :Integer
- 2050      2  TRANSMITTED        :Integer ValueP
- 2056      2  FIELD_BASE        :Integer ValueP
- 2072      2  REDUCED_ELT        :Integer
- 2070      2  REDUCING_ROW       :Integer

```

396

397

398

399

400

```

401      ( the ENCODE procedure.  It accepts the number of channels
402      transmitted, the number of channels to be received, the
403      modulus, and the field_base.  It then generates an encoding
404      key and begins reading plaintext words.  It encodes the
405      plaintext words and prints them out (transmits them)
406      until it encounters an end-of-file flag.
407

```

```

20 408      procedure ENCODE ( transmitted : integer;
20 409      received       : integer;
20 410      modulus        : integer;
20 411      field_base     : integer;
20 412      output_channel : channel_array );

```

ENCODE

```

05 IC Line# Source Line IBM Personal Computer Pascal Compiler V1.00
      413
20    414 var enf           : matrix;
20    415     enf_prime      : matrix;
20    416     submatrix      : matrix;
20    417     encode_key     : matrix;
20    418     decode_key     : matrix;
20    419     cool_decoder   : matrix;
20    420     index           : integer;
20    421     key_column      : integer;
20    422     row             : integer;
20    423     column          : integer;
20    424     dimension       : integer;
20    425     EDT             : boolean;
20    426     response       : char;
      427
      428
20    429 begin
      430
21    431     dimension := TWO_TO_THE (field_base);
      432
21    433     BUILD_ENF (enf, transmitted, modulus, field_base);
21    434     TRANSPOSE(enf,enf_prime,dimension,(dimension - 2));
21    435     writeln;writeln('ENF MATRIX----->');
21    436     for row := 1 to (dimension - 2) do begin
22    437         writeln;
22    438         for column := 1 to dimension do
22    439             WRITE_OCTAL (enf_prime[row][column], field_base)
21    440         end;
21    441         page;
21    442     EXTRACT_SUBMATRIX (submatrix, enf, transmitted, received,field
21    442     d_base);
21    443     writeln;writeln('SUBMATRIX----->');
21    444     for row := 1 to (transmitted - received) do begin
22    445         writeln;
22    446         for column := 1 to transmitted do
22    447             WRITE_OCTAL (submatrix[row][column], field_base)
21    448         end;
21    449         page;
21    450     BUILD_ENCODE_KEY (encode_key, submatrix, transmitted, received,
21    450     d,
21    451         modulus, field_base);
21    452     writeln;writeln('ENCODE KEY----->');
21    453     for row := 1 to (transmitted - received) do begin
22    454         writeln;
22    455         for column := 1 to transmitted do
22    456             WRITE_OCTAL (encode_key[row][column], field_base)

```

ENCODE

```

30 10  Line#   Source Line      IBM Personal Computer Pascal Compiler V1.00
21   457           end;
21   458           page;
    459
    460   ( the encode routine now reads in "received" cleartext words,
    461     generates "transmitted" - "received" coded words, and sends
    462     all "transmitted" words out.
    463   )
21   464   EOT := FALSE;
21   465   repeat
12   466     begin
23   467       writeln('please enter, on one line, in octal and separate
23   467       d');
23   468       writeln('by blanks, the values to be transmitted over the
23   468       ');
23   469       writeln('transmitters ',received:2,' channels');
23   470       for index := 1 to received do begin
24   471         READ_OCTAL (output_channel[index]);
23   472       end;
    473
23   474       writeln('words transmitted are (in channel order):');
    475
23   476       for index := (received+1) to transmitted do begin
24   477         output_channel[index] := 0;
24   478         for key_column := 1 to received do
24   479           output_channel[index] :=
24   480             ADD ( output_channel[index],
24   481                 MULTIPLY (output_channel[key_column],
24   482                           encode_key[(transmitted-index)+1]
24   483                             [key_column],
24   484                             modulus, field_base)
24   484       )
23   485       end;
    486
23   487       for index := 1 to transmitted do
23   488         WRITE_OCTAL (output_channel[index], field_base);
23   489       writeln;writeln;
    490
23   491       writeln('do you want to send another ',received:2,' words
23   491       ');
23   492       writeln('(type y or n)');
23   493       readln(response);
23   494       if (response = 'n') then EOT := TRUE;
    495
23   496     end
21   497   until (EOT);
21   498   page

```

ENCODE

```

10 10 Line# Source Line IBM Personal Computer Pascal Compiler V1.00
      499
      500 end;

Symtab 500 Offset Length Variable - ENCODE
      - 72 12400 Return offset, Frame length
      - 0 2 TRANSMITTED :Integer ValueP
      - 2 2 RECEIVED :Integer ValueP
      - 4 2 MODULUS :Integer ValueP
      - 2122 2048 ENF :Array
      -12364 2 INDEX :Integer
      -12368 2 ROW :Integer
      -12374 1 EOT :Boolean
      -12370 2 COLUMN :Integer
      - 6 2 FIELD_BASE :Integer ValueP
      - 4170 2048 ENF_PRIME :Array
      -12372 2 DIMENSION :Integer
      -12376 1 RESPONSE :Char
      - 6218 2048 SUBMATRIX :Array
      - 8266 2048 ENCODE_KEY :Array
      -10314 2048 DECODE_KEY :Array
      -12366 2 KEY_COLUMN :Integer
      - 70 64 OUTPUT_CHANNEL :Array ValueP
      -12362 2048 COOL_DECODER :Array

501
502
503 ( THE MAIN ROUTINE. THIS CODE READS IN THE NUMBER OF
504 CHANNELS SENT AND THE NUMBER OF CHANNELS WHICH NEED
505 TO BE RECIEVED, AND GENERATES AN ENCODE-NORMAL FORM
506 MATRIX FOR THAT CHOICE OF 'TRANSMITTED' AND 'RECIEVED'. )
507
508
10 509 var field_base : integer;
10 510 modulus : integer;
10 511 transmitted : integer;
10 512 received : integer;
10 513 channels : channel_array;
10 514 index : integer;
515
516
517
10 518 begin
519
11 520 writeln(chr(27),'M'); (enable elite type on printer)
521
11 522 writeln('please enter, on one line and separated by blanks,')

```

END

```

06 IC  Line#   Source Line      IBM Personal Computer Pascal Compiler V1.00
11      522      ;
11      523      writeln('the field-base, modulus, number of channels to be se
11      523      nt,');
11      524      writeln('and number of channels to be received.  The modulus'
11      524      ');
11      525      writeln('should be an octal number; all other numbers should'
11      525      ');
11      526      writeln('be decimal. ');
11      527      writeln;
11      528
11      529      read (field_base);
11      530      READ_OCTAL (modulus);
11      531      modulus := modulus - TWO_TO_THE(field_base);
11      532      read (transmitted);
11      533      readln (received);
11      534      writeln;
11      535
11      536      writeln('thank you...please wait');writeln;
11      537
11      538      ENCODE (transmitted, received, modulus, field_base, channels);
11      538      ;
11      539
11      540      writeln(chr(27),chr(64));          (disable special print modes
11      540      )
11      541
00      542      end.

```

Symtab	542	Offset	Length	Variable	
		0	76	Return offset, Frame length	
		74	2	INDEX	:Integer Static
		4	2	MODULUS	:Integer Static
		10	64	CHANNELS	:Array Static
		8	2	RECEIVED	:Integer Static
		2	2	FIELD_BASE	:Integer Static
		6	2	TRANSMITTED	:Integer Static

Errors Warns In Pass One

0 0

Appendix I

Program for decoding procedure
(including Stages 1, 2, 3 and 4)

```

10 10 Line#   Source Line      IBM Personal Computer Pascal Compiler V1.00
20      1   program dnf (input,output);
      2
10      3   const  zero      = 0;
10      4           one      = 1;
10      5           maxindex  = 32;
10      6           WORDLENGTH = 16;
      7
10      8   type   mat_row    = array [1..maxindex] of integer;
10      9           matrix    = array [1..maxindex] of mat_row;
      10
10     11   var    index      : integer;
10     12           van       : matrix;
10     13           dnf       : matrix;
10     14           dnf_prime  : matrix;
10     15           received   : integer;
10     16           transmitted : integer;
10     17           field_base  : integer;
10     18           modulus    : integer;
10     19           row        : integer;
10     20           col        : integer;
10     21           rows       : integer;
10     22           datarow    : integer;
10     23           datarows   : integer;
10     24           extra_desid : integer;
10     25           columns   : integer;
10     26           dimension  : integer;
10     27           temp       : integer;
10     28           channel    : integer;
10     29           desired_channels : integer;
10     30           reducing_elt  : integer;
10     31           reduced_elt  : integer;
10     32           dead_channels : integer;
10     33           dead_channel  : mat_row;
10     34           decoder     : matrix;
10     35           desired_channel : mat_row;
10     36           data          : matrix;
10     37           desiderata   : matrix;
10     38           active_channel : mat_row;
10     39           codeword      : mat_row;
10     40           clearword    : integer;
10     41           continue    : char;
10     42           active      : boolean;
10     43           EOT         : boolean;      (End Of Transmission
10     44
10     45

```

Page 2
05-23-84
17:43:58

```

10 10 Line# Source Line IBM Personal Computer Pascal Compiler V1.00
    46 ( the TWO_TO_THE function makes up for the lack of a generalize
    46 d
    47 exponentiation operator in standard Pascal. It returns two
    48 raised to the power of its caller-supplied argument.
    48 )
    49
20 50 function TWO_TO_THE (argument : integer): integer;
    51
20 52 var accumulator : integer;
20 53 index : integer;
    54
20 55 begin
    56 accumulator := 1;
    57 for index := 1 to argument do
    58 accumulator := accumulator * 2;
+ 21 59 TWO_TO_THE := accumulator;
    10 60 end;

Symbtab 60 Offset Length Variable - TWO_TO_THE
- 2 14 Return offset, Frame length
- 6 2 (function return) : Integer
- 0 2 ARGUMENT :Integer ValueP
- 10 2 INDEX :Integer
- 8 2 ACCUMULATOR :Integer

    61
    62
    63
    64 ( the READ_OCTAL routine; this routine allows the user of
    65 the program to input his values in octal rather than in
    66 decimal; it replaces the Pascal standard "read" routine. )
    67
20 68 procedure READ_OCTAL (var total : integer);
    69
20 70 const BLANK = ' ';
    71
    72 var inchar : char;
    73
    74
20 75 begin
    76 read (inchar);
    77 total := 0;
    78
    79 while (inchar = BLANK) do
    80 read (inchar);
    81

```

READ_OCTAL

Page 3
05-23-84
17:44:02

IBM Personal Computer Pascal Compiler V1.00

```

10 10 Line# Source Line
21 82 while not (inchar = BLANK) do begin
22 83 total := total * 8 + (ord(inchar) - ord('0'));
22 84 read (inchar)
22 85 end
22 86
10 87 end;
```

```

tab 87 Offset Length Variable - READ_OCTAL
- 2 8 Return offset, Frame length
- 0 2 TOTAL :Integer VarP
- 6 1 INCHAR :Char
```

```

88
89
90 (the WRITE_OCTAL routine; it replaces the Pascal standard
91 "write" routine and allows the program to report its
92 output values in octal rather than in decimal. )
93
20 94 procedure WRITE_OCTAL (number : integer;
20 95 field_base : integer );
96
20 97 var outbuf : array [1..WORDLENGTH] of char;
20 98 temp : integer;
20 99 index : integer;
100
101
20 102 begin
21 103 for index := 1 to WORDLENGTH do outbuf[index] := '0';
21 104 index := 1;
105
21 106 while (number > 0) do begin
22 107 temp := number mod 8;
22 108 outbuf[index] := chr (ord('0') + temp);
22 109 index := index + 1;
22 110 number := number div 8
21 111 end;
112
21 113 temp := ((field_base + 2) div 3);
21 114 if (temp < 1) then temp := 1;
21 115 if (temp < (index - 1)) then temp := index - 1;
21 116 for index := temp downto 1 do write(outbuf[index]);
21 117 write(' ');
118
10 119 end;
```

```

tab 119 Offset Length Variable - WRITE_OCTAL
```

WRITE_OCTAL

Page 4
05-23-84
17:44:05

```

JG IC Line# Source Line IBM Personal Computer Pascal Compiler V1.00
- 4 30 Return offset, Frame length
- 0 2 NUMBER :Integer ValueP
- 24 2 TEMP :Integer
- 26 2 INDEX :Integer
- 2 2 FIELD_BASE :Integer ValueP
- 22 16 OUTBUF :Array

120
121
122
123 ( The ADD function returns the logical xor of its two caller-
124 supplied arguments. This is addition over GF(n) for any n. )
125
20 126 function ADD (term1 : integer;
20 127 term2 : integer ): integer;
128
20 129 begin
= 21 130 ADD := ( (term1 or term2) and (not(term1 and term2)) )
10 131 end;

Syntab 131 Offset Length Variable - ADD
- 4 10 Return offset, Frame length
- 8 2 (function return) : Integer
- 0 2 TERM1 :Integer ValueP
- 2 2 TERM2 :Integer ValueP

132
133
134 ( The MULTIPLY function performs multiplication over GF(n)
135 modulo the caller-supplied modulus and returns the result
136 of the multiplication. )
137
20 138 function MULTIPLY (factor1 : integer;
20 139 factor2 : integer;
20 140 modulus : integer;
20 141 field_base : integer ): integer;
142
20 143 var index : integer;
20 144 answer : integer;
145
20 146 begin
147
21 148 answer := 0;
149
21 150 for index := 0 to (field_base - 1) do
21 151 begin

```

MULTIPLY

Page 5
05-23-84
17:44:08

```

16 10 Line# Source Line IBM Personal Computer Pascal Compiler V1.00
22 152 answer := answer * 2;
22 153
22 154 if (( (factor1 mod TWO_TO_THE (field_base - index))
22 155 div TWO_TO_THE (field_base - (index+1)) ) > 0)
22 156 then answer := ADD (answer, factor2);
22 157
22 158 if ( (answer div TWO_TO_THE (field_base)) > 0)
22 159 then answer := ADD (answer,
22 160 TWO_TO_THE (field_base) + modulus)
21 161 end;
= 21 162 MULTIPLY := answer
10 163 end;

```

```

Symbol 163 Offset Length Variable - MULTIPLY
- 5 20 Return offset, Frame length
- 12 2 (function return) : Integer
- 0 2 FACTOR1 :Integer ValueF
- 14 2 INDEX :Integer
- 16 2 ANSWER :Integer
- 2 2 FACTOR2 :Integer ValueF
- 4 2 MODULUS :Integer ValueF
- 6 2 FIELD_BASE :Integer ValueF

```

```

164
165
166 { The INVERSE function. It accepts a field element and
167 returns the element's multiplicative inverse. This
168 implementation is very slow & primitive-- it should be
169 replaced by Davida's inverse routine or some other fast
170 implementation at the first opportunity. }
171

```

```

20 172 function INVERSE ( element      : integer;
20 173                   field_base   : integer;
20 174                   modulus      : integer ): integer;
20 175
20 176 var   index      : integer;
20 177       answer     : integer;
20 178       squares    : integer;
20 179
20 180 begin
20 181
21 182     answer := 1;
21 183     squares := element;
20 184
21 185     for index := 1 to (field_base - 1) do
21 186         begin

```

INVERSE

Page 6
05-23-84
17:44:11

```

IBM Personal Computer Pascal Compiler V1.00
  22 187      squares := MULTIPLY (squares,squares,modulus,field_base);
  22 188      answer  := MULTIPLY (answer,squares,modulus,field_base)
  21 189      end;
      190
= 21 191      INVERSE := answer
      192
  10 193  end;

```

```

Symtab 193  Offset Length  Variable - INVERSE
      -    6      20  Return offset, Frame length
      -   10       2  (function return) : Integer
      -    0       2  ELEMENT :Integer ValueF
      -   12       2  INDEX :Integer
      -   14       2  ANSWER :Integer
      -    2       2  FIELD_BASE :Integer ValueF
      -    4       2  MODULUS :Integer ValueF
      -   16       2  SQUARES :Integer

```

```

194
195 { The DIVIDE function performs Galois-field division.
196   it accepts dividend, divisor, modulus, and field-base
197   (in that order), takes the inverse of the divisor, and
198   multiplies the result by the dividend.
199 }

```

```

20 200 function DIVIDE ( dividend      : integer;
20 201                  divisor       : integer;
20 202                  modulus       : integer;
20 203                  field_base    : integer ): integer;
204
20 205 var divisor_inverse : integer;
206
20 207 begin
208
21 209     divisor_inverse := INVERSE (divisor, field_base, modulus);
= 21 210     DIVIDE := MULTIPLY (dividend, divisor_inverse,
21 211                       modulus, field_base
212                       )
  10 213 end;

```

```

Symtab 213  Offset Length  Variable - DIVIDE
      -    8      16  Return offset, Frame length
      -   12       2  (function return) : Integer
      -    0       2  DIVIDEND :Integer ValueF
      -    2       2  DIVISOR :Integer ValueF
      -    4       2  MODULUS :Integer ValueF
      -    6       2  FIELD_BASE :Integer ValueF

```

DIVIDE

```

26 10  Line#   Source Line      IBM Personal Computer Pascal Compiler V1.00
-    14      2  DIVISOR_INVERSE      : Integer

214
215
216  ( The HERMITE_NORMALIZE routine takes a matrix which is
217    at least two columns wide and which is also at least
218    as tall as it is wide and reduces it to Hermite normal
219    form (i.e. to a form with an identity matrix at the top.) )
220
20 221  procedure HERMITE_NORMALIZE  (var m          : matrix;
20 222                                rows          : integer;
20 223                                cols          : integer;
20 224                                mod1         : integer;
20 225                                f_base       : integer );
20 226
20 227  var row          : integer;
20 228      col          : integer;
20 229      reducing_col  : integer;
20 230      reducing_elt  : integer;
20 231      reduced_elt   : integer;
20 232      index         : integer;
20 233      temp          : integer;
20 234
20 235  begin
20 236
21 237      if (cols < 2) then
21 238          writeln ('stripped matrix has <2 cols: error')
21 239      else begin
20 240
22 241          for reducing_col := 1 to cols do begin
23 242              index := reducing_col;
23 243              while ( (m [reducing_col][index] = 0) and
23 244                  (index < reducing_col) ) do
23 245                  index := index + 1;
23 246
23 247              if (not(index = reducing_col)) then      (switch c
23 247  ols)
23 248                  for row := 1 to rows do begin
24 249                      temp := m [row][reducing_col];
24 250                      m [row][reducing_col] := m [row][index];
24 251                      m [row][index] := temp
23 252                  end;
23 253
23 254                  reducing_elt := m [reducing_col][reducing_col];
23 255
23 256                  (set leading elts. of columns to 1 by dividing cols by constant
HERMITE_NORMALIZE

```

Page 8
05-23-84
17:44:17

```

16 10  Line#   Source Line      IBM Personal Computer Pascal Compiler V1.00
      256      .)
      257
23   258      temp := reducing_elt;
23   259      if (not (temp = 1) ) then begin
24   260          m[reducing_col][reducing_col] := 1;
24   261          for row := (reducing_col + 1) to rows do
24   262              m [row][reducing_col] := DIVIDE (m[row][reducing_col]
24   262      , temp,
24   263                      mod1, f_base      )
23   264      end;
      265
      266      {column-reduce by clearing row 'reducing-col' using entry
      267      m[reducing_col][reducing_col].
      268
23   269      for col := 1 to cols do
      270
23   271          if (not(col = reducing_col)) then begin
24   272              reduced_elt := m [reducing_col][col];
24   273              if (not(reduced_elt = 0)) then
24   274                  for row := reducing_col to rows do
24   275                      m [row][col] :=
24   276                      ADD (m [row][col],
24   277                          MULTIPLY (m [row][reducing_col],
24   278                              reduced_elt, mod1, f_base ) )
24   279              end
      280
23   281          end
22   282      end
      283
10   284  end;

```

Symbol	Offset	Length	Variable - HERMITE_NORMALIZE	
	- 10	42	Return offset, Frame length	
	- 0	2	M	:Array VarP
	- 2	2	ROWS	:Integer ValueP
	- 4	2	COLS	:Integer ValueP
	- 14	2	ROW	:Integer
	- 16	2	COL	:Integer
	- 6	2	MODL	:Integer ValueP
	- 8	2	F_BASE	:Integer ValueP
	- 24	2	INDEX	:Integer
	- 26	2	TEMP	:Integer
	- 18	2	REDUCING_COL	:Integer
	- 22	2	REDUCED_ELT	:Integer
	- 20	2	REDUCING_ELT	:Integer

HERMITE_NORMALIZE

```

13 10 Line# Source Line IBM Personal Computer Pascal Compiler V1.00
    285
    286
    287
    288
    289 ( The CONSTRUCT_VAN routine. This procedure constructs a
    290 square vandermonde matrix with the dimension supplied by the
    291 calling routine.
    292
    293 procedure CONSTRUCT_VAN (var van          : matrix;
    294                          n                : integer;
    295                          field_base       : integer;
    296                          modulus         : integer );
    297
    298 var   row          : integer;
    299       column       : integer;
    300       exponent     : integer;
    301       index        : integer;
    302       temp         : integer;
    303
    304 begin
    305
    306   if (n < 3) then writeln ('van dimension < 3: error')
    307   else
    308     begin
    309
    310       (build first row of van. )
    311
    312       van [1][1] := 1;
    313       for column := 2 to n do
    314         van [1][column] := 0;
    315
    316       (build second row of van. )
    317
    318       for column := 1 to n do
    319         van [2][column] := 1;
    320
    321       (build third row of van. )
    322
    323       van [3][1] := 1;
    324       van [3][2] := 2;
    325       for column := 3 to n do
    326         van [3][column] :=
    327           MULTIPLY (van [3][column - 1], 2,
    328                     modulus, field_base
    329                   );
    330       (build remaining rows of van.)
    331
CONSTRUCT_VAN

```

```

16 10 Line# Source Line IBM Personal Computer Pascal Compiler V1.00
      331
22 332      if (n > 3) then
22 333          for row := 4 to n do begin
23 334              van [row][1] := 1;
23 335              for column := 2 to n do
23 336                  van [row][column] :=
23 337                      MULTIPLY (van [row - 1][column], van [1][column
23 337      1,
23 338                      modulus, field_base
23 338      )
23 339      end
      340
      341
22 342      end
10 343  end;

```

Offset	Length	Variable	CONSTRUCT_VAN
- 8	32	Return offset, Frame length	
- 0	2	VAN	:Array VarP
- 2	2	N	:Integer ValueP
- 12	2	ROW	:Integer
- 4	2	FIELD_BASE	:Integer ValueP
- 6	2	MODULUS	:Integer ValueP
- 14	2	COLUMN	:Integer
- 18	2	INDEX	:Integer
- 20	2	TEMP	:Integer
- 16	2	EXPONENT	:Integer

```

344
345
346
347      ( the TRANSPOSE routine accepts a matrix and its dimensions
348        and produces the transpose of the matrix. )
349
20 350  procedure TRANSPOSE (      m      : matrix;
20 351                          var m_prime : matrix;
20 352                          m_rows   : integer;
20 353                          m_cols   : integer );
20 354
20 355  var row : integer;
20 356      col : integer;
20 357
20 358  begin
21 359      for row := 1 to m_rows do
21 360          for col := 1 to m_cols do
21 361              m_prime[col][row] := m[row][col];

```

TRANSPOSE

```

JG IC  Line#  Source Line      IBM Personal Computer Pascal Compiler V1.00
10      362      end;

Svmtab  362      Offset Length  Variable - TRANSPOSE
          - 2054    2066      Return offset, Frame length
          - 2046    2048      M                      :Array    ValueP
          - 2048      2      M_PRIME                  :Array    VarP
          - 2050      2      M_ROWS                    :Integer  ValueP
          - 2052      2      M_COLS                    :Integer  ValueP
          - 2058      2      ROW                      :Integer
          - 2060      2      COL                      :Integer

363
364
365
366      ( THE MAIN ROUTINE.  THIS CODE GOES THROUGH THE ENTIRE
367        DECODING PROCESS, WHICH IS BROKEN INTO COLD, COOL AND
368        HOT PRECOMPUTE STAGES AND ONLINE DECODE STAGE.
369
10      370      begin
371
372      ( COLD PRECOMPUTE STAGE BEGINS HERE.
373
374      ( First, we read in the modulus and field-base for the
375        Galois field to be used in our calculations.
376
11      377      writeln('Please enter, on one line and separated by a blank,
11      377      ');
11      378      writeln('the field-base and modulus to be used.  The');
11      379      writeln('field-base should be a decimal number and the');
11      380      writeln('modulus should be an octal number. ');
11      381
11      382      read(field_base);
11      383      READ_OCTAL(modulus);
11      384
11      385      modulus := modulus - TWO_TO_THE(field_base);
11      386
11      387
11      388      ( Next, we construct a Vandermonde matrix called VAN.
11      389
11      390      dimension := TWO_TO_THE(field_base);
11      391
11      392      CONSTRUCT_VAN (van, dimension, field_base, modulus);
11      393
11      394
11      395      ( COOL PRECOMPUTE STAGE BEGINS HERE.
11      396

```

```

06 IC Line# Source Line      IBM Personal Computer Pascal Compiler V1.00
      397      { Next, we read in the number of channels to be sent
      398          by the transmitting node.
      399      }
11     400      writeln('Please enter the number of channels to be sent');
11     401      writeln('by the transmitting node. This should be a');
11     402      writeln('decimal number. ');
      403
11     404      readln(transmitted);
      405
      406
      407      { Next, we read the number of channels to be received by
      408          the receiving node.
      409      }
11     410      writeln('please enter the number of channels active at');
11     411      writeln('the receivers node; this should be a decimal number.
11     411      ');
      412
11     413      readln(received);
      414
      415
      416      { Now we strip away the extraneous rows and columns of the
      417          vandermonde matrix. We leave only the topmost n rows and
      418          the leftmost k columns of VAN.
      419      }
11     420      rows := transmitted;
11     421      columns := received;
      422
      423
      424      { Next, we hermite-normalize van to give us a tall, thin matrix
      425          with an identity at the top.
      426      }
11     427      HERMITE_NORMALIZE (van, rows, columns, modulus, field_base);
      428
      429
      430      { Finally, we construct our "special" left-kernel for the
      431          stripped, col-reduced VAN. This matrix is short and
      432          fat, with an identity at the left, and it is our DNF
      433          (Decode-Normal-Form) matrix.
      434      }
11     435      for row := 1 to (transmitted - received) do
11     436          for col := 1 to received do
11     437              dnf [row][col] := van [row + received][col];
      438
11     439      for row := 1 to (transmitted - received) do
11     440          for col := (received + 1) to transmitted do
11     441              if ( (col - received) = row) then

```

DNF

Page 13
05-23-84
17:44:40

```

05 IC Line# Source Line IBM Personal Computer Pascal Compiler V1.00
11 442 dnf [row][col] := 1
11 443 else
11 444 dnf [row][col] := 0;
11 445
11 446 TRANSPOSE (dnf, dnf_prime, (transmitted - received), transmit
11 446 ted);
11 447
11 448 HERMITE_NORMALIZE (dnf_prime, transmitted, (transmitted - rec
11 448 eived),
11 449 modulus, field_base);
11 450
11 451 TRANSPOSE (dnf_prime, dnf, transmitted, (transmitted - receiv
11 451 ed));
11 452
11 453
11 454 writeln;
11 455 writeln ('DNF matrix for ',received:2,' out of ',transmitted:
11 455 2);
11 456 write ('channels over GF 2**(',field_base:1,') mod ');
11 457 temp := modulus + TWO_TO_THE (field_base);
11 458 WRITE_OCTAL(temp, field_base);
11 459 writeln('is: ');writeln;
11 460 for row := 1 to (transmitted - received) do begin
12 461 writeln;
12 462 for col := 1 to transmitted do
12 463 WRITE_OCTAL (dnf [row][col], field_base);
11 464 end;
11 465 writeln;writeln;
11 466
11 467 ( HOT PRECOMPUTE STAGE BEGINS HERE. )
11 468
11 469 writeln('please enter, on one line and separated by blanks,')
11 469 ;
11 470 writeln('the numbers of the ',received:2,' channels active');
11 471 writeln('at the receiving node. These numbers should be deci
11 471 mal. ');
11 472
11 473 for index := 1 to (received - 1) do read (active_channel [ind
11 473 ex]);
11 474 readln (active_channel [received]);
11 475
11 476
11 477 ( Here we fill up the data matrix. )
11 478
11 479 row := 1;
11 480 for index := 1 to received do

```

DNF

```

36 10  Line#  Source Line      IBM Personal Computer Pascal Compiler V1.00
11   481      if (active_channel [index] <= (transmitted - received)) the
11   481      n begin
12   482          for col := 1 to transmitted do
12   483              data [row][col] :=
12   484                  dnf [ active_channel [index] ][col];
12   485              row := row + 1
11   486          end;
11   487      datarows := row - 1;
11   488
11   489
11   490      ( Here we fill up the desiderata matrix and those rows of the
11   491          decoder matrix corresponding to channels which we are receivin
11   491          g.)
11   492
11   493      desired_channels := 0;
11   494      dead_channels := 0;
11   495      extra_desid := 1;
11   496      for channel := 1 to transmitted do begin
12   497          active := FALSE;
12   498          for index := 1 to received do
12   499              if (active_channel [index] = channel) then active := TRUE
12   499          ;
12   500          if ( (not active) and (channel <= received) ) then begin
13   501              desired_channels := desired_channels + 1;
13   502              desired_channel [desired_channels] := channel;
13   503              if (channel > (transmitted - received)) then begin
14   504                  dead_channels := dead_channels + 1;
14   505                  dead_channel [dead_channels] := channel;
14   506                  for col := 1 to transmitted do
14   507                      desiderata [channel][col] := data [extra_desid][col];
14   508                  extra_desid := extra_desid + 1
14   509              end
13   510              else
13   511                  for col := 1 to transmitted do
13   512                      desiderata [channel][col] := dnf [channel][col]
13   513              end
12   514          else if (not active) then begin
13   515              dead_channels := dead_channels + 1;
13   516              dead_channel [dead_channels] := channel
13   517          end
12   518          else if (channel <= received) then
12   519              for col := 1 to transmitted do
12   520                  if (col = channel) then decoder [channel][col] := 1
12   521                  else decoder [channel][col] := 0
11   522          end;
11   523

```

DNF

Page 15
05-23-84
17:44:51

```

06 IC Line# Source Line IBM Personal Computer Pascal Compiler V1.00
      524
      525 { Here we clear the columns in the desiderata matrix correspond
      525 ing
      526 to channels which we are not receiving.
      526 }
      527
11 528 row := 1;
11 529 for index := 1 to dead_channels do begin
12 530 if (dead_channel [index] > (transmitted - received)) then b
12 530 egin
13 531 for channel := 1 to desired_channels do
13 532 if (not (desired_channel[channel] = dead_channel[index])) th
13 532 en begin
14 533 reducing_elt := data [row][dead_channel[index]];
14 534 reduced_elt := desiderata [ desired_channel[channel] ]
14 535 [dead_channel[index]];
14 536 for col := 1 to transmitted do
14 537 desiderata [ desired_channel[channel] ] [col] :=
14 538 ADD ( MULTIPLY(desiderata [desired_channel[channel]] [
14 538 col],
14 539 reducing_elt, modulus, field_base),
14 540 MULTIPLY(data[row][col], reduced_elt,
14 541 modulus, field_base))
13 542 end;
13 543 for datarow := (row + 1) to datarows do begin
14 544 reducing_elt := data[row][dead_channel[index]];
14 545 reduced_elt := data [datarow][dead_channel[index]];
14 546 for col := 1 to transmitted do
14 547 data [datarow][col] :=
14 548 ADD ( MULTIPLY(data[datarow][col], reducing_elt,
14 549 modulus, field_base),
14 550 MULTIPLY(data[row][col], reduced_elt,
14 551 modulus, field_base))
13 552 end;
13 553 row := row + 1
13 554 end
11 555 end;
      556
      557
      558 { Here we obtain ones in the "lead" columns of the desiderata
      559 rows by dividing through by the values previously in those
      560 columns.
      561 }
11 562 for channel := 1 to desired_channels do begin
12 563 reducing_elt := desiderata [desired_channel[channel]]
12 564 [desired_channel[channel]];
```

DNF

Page 16
05-23-84
17:44:59

```

JG IC Line# Source Line      ISM Personal Computer Pascal Compiler V1.00
12 565      for col := 1 to transmitted do
12 566      desiderata [desired_channel[channel]][col] :=
12 567      DIVIDE (desiderata [desired_channel[channel]][col],
12 568      reducing_elt, modulus, field_base);
11 569      end;
12 570
12 571
12 572      ( Now fill up the rows of the decoder matrix corresponding
12 573      to channels which were desired but not active. )
12 574
11 575      for channel := 1 to desired_channels do
11 576      for col := 1 to transmitted do
11 577      decoder [desired_channel[channel]][col] :=
11 578      desiderata [desired_channel[channel]][col];
12 579
12 580
12 581      ( Now we print out the decoder matrix. )
12 582
11 583      writeln('Decoder matrix for the active channels listed above
11 583      is:');
12 584
11 585      writeln:
11 586      for row := 1 to received do begin
12 587      for col := 1 to transmitted do
12 588      WRITE_OCTAL ( decoder [row][col], field_base);
12 589      writeln
11 590      end;
11 591      writeln;
12 592
12 593
12 594      ( DECODING BEGINS HERE. )
12 595
11 596      EOT := FALSE;
11 597      while (not EOT) do begin
12 598
12 599      writeln('please enter, on one line and separated by blanks,')
12 600      ;
12 601      writeln('the data received on each of the channels active at')
12 601      );
12 602      writeln('the receivers node. The data should be in the form')
12 602      );
12 603      writeln('of octal numbers, and should be entered in order of')
12 603      );
12 604      writeln('increasing channel number. ');
12 605      for index := 1 to transmitted do

```

END

```

15 10 Line# Source Line IBM Personal Computer Pascal Compiler V1.00
12 606 codeword [index] := 0;
12 607
12 608 for index := 1 to received do
12 609 READ_OCTAL ( codeword [ active_channel [index] ] );
12 610
12 611 writeln;
12 612 writeln('the ',received:2,' transmitted cleartext words were'
12 612 );
12 613 writeln('(octal numbers expressed in channel order):');
12 614 writeln;
12 615
12 616 for index := 1 to received do begin
13 617 clearword := 0;
13 618 for col := 1 to transmitted do
13 619 clearword := ADD (clearword,
13 620 MULTIPLY (decoder[index][col],
13 621 codeword[col],
13 622 modulus, field_base) );
13 623 WRITE_OCTAL (clearword, field_base)
12 624 end;
12 625 writeln;
12 626
12 627 writeln('do you want to decode another ',received:2,' words?'
12 627 );
12 628 writeln('(type y or n).');
12 629
12 630 readln(continue);
12 631 if (continue = 'n') then EOT := TRUE
12 632
12 633 end
12 634
00 635 end.

```

Symtab	Offset	Length	Variable	
	0	12592	Return offset, Frame length	
	2052	2048	DNF	:Array Static
	6158	2	COL	:Integer Static
	12590	1	EOT	:Boolean Static
	4	2048	VAN	:Array Static
	6156	2	ROW	:Integer Static
	6160	2	ROWS	:Integer Static
	8360	2048	DATA	:Array Static
	6172	2	TEMP	:Integer Static
	2	2	INDEX	:Integer Static
	12588	1	ACTIVE	:Boolean Static
	6162	2	DATAROW	:Integer Static

Page 18

05-23-84

17:45:13

JS IC	Line#	Source Line	IBM Personal Computer Pascal Compiler V1.00
		6168 2	COLUMNS :Integer Static
		6174 2	CHANNEL :Integer Static
		6248 2048	DECODER :Array Static
		6154 2	MODULUS :Integer Static
		6164 2	DATAROWS :Integer Static
		12520 64	CODEWORD :Array Static
		12586 1	CONTINUE :Char Static
		4100 2048	DNF_PRIME :Array Static
		6148 2	RECEIVED :Integer Static
		6170 2	DIMENSION :Integer Static
		12584 2	CLEARWORD :Integer Static
		6152 2	FIELD_BASE :Integer Static
		10408 2048	DESIDERATA :Array Static
		6150 2	TRANSMITTED :Integer Static
		6166 2	EXTRA_DESID :Integer Static
		6180 2	REDUCED_ELT :Integer Static
		6176 2	DESIRED_CHANNELS :Integer Static
		6178 2	REDUCING_ELT :Integer Static
		6184 64	DEAD_CHANNEL :Array Static
		6182 2	DEAD_CHANNELS :Integer Static
		8296 64	DESIRED_CHANNEL :Array Static
		12456 64	ACTIVE_CHANNEL :Array Static

Errors	Warns	In Pass One
0	0	